robotron

Programmtechnische
Beschreibung

BASIC-Interpreter

C 1015-0300-1 M 3030

Arbeitsplatzcomputer A 7100 Betriebssystem SCP1700

Programmtechnische Beschreibung

BASIC-Interpreter

AC A7100

VEB Robotron-Projekt Dresden

C 1015-0300-1 M 3030

Die vorliegende Systemunterlagendokumentation, Programmtechnische Beschreibung, BASIC-Interpreter, entspricht dem Stand von 6/86.

Nachdruck, jegliche Vervielfaeltigung oder Auszuege daraus sind unzulaessig.

Die Ausarbeitung erfolgte durch ein Kollektiv des VEB Robotron-Projekt-Dresden.

Herausgeber:

VEB Robotron-Projekt Dresden
8010 Dresden, Leningrader Strasse 9

(C) 1986

Kurzreferat

In der vorliegenden Schrift wird die BASIC-Implementierung fuer das Betriebssystem SCP auf dem Mikrorechner AC A 7100 beschrieben. Die Schrift enthaelt die Beschreibung aller sprachlichen Konstruktionen sowie alle Informationen zur Bedienung des BASIC-Systems.

Ag 706/297/87 III/12/12

Inhaltsverzeichnis

		Seite
1.	Allgemeine Informationen	7
1.1.	Beschreibungsform der Sprache	7
1.2.	Start des Interpreters	7
1.3.	Operationsmodi	9
1.4.	Eingabe von Programmen	9
1.5.	Fehlermeldungen	9
2.	Programmstruktur	10
2.1.	Zeilenformat	10
2.2.	Zeilennummern	10
3.	Grundelemente	11
3.1.	Zeichensatz	11
3.2.	Steuerzeichen	12
3.3.	Konstanten	12
3.4.	Variable	14
3.4.1.	Variablennamen und Deklarationszeichen	14
3.4.2.	Feldvariable	15
3.4.3.	Speicherplatzbedarf	15
3.5.	Typ-Konvertierungen	16
4.	Ausdruecke und Operatoren	18
4.1.	Allgemeines	18
4.2.	Arithmetische Operatoren	18
4.3.	Vergleichsoperatoren	20
4.4.	Logische Operatoren	21
4.5.	Funktionale Operatoren	23
4.6.	String-Ausdruecke und String-Vergleich	23
5.	Kommandos	24
5.1.	Allgemeines	24
5.2.	Kommandos zur Programmaufbereitung	24
5.2.1.	AUTO	24
5.2.2.	DELETE	25
5.2.3.	EDIT	25
5.2.4.	LIST	28
5.2.5.	LLIST	29
5.2.6.	RENUM	29
5.2.7.	NEW	30
5.3.	Programmausfuehrungskommandos	30
5.3.1.	CLEAR	30
5.3.2.	CONT	31
5.3.3.	RUN	31
5.3.4.	TRON/TROFF	32
5.3.5.	System	33
5.4.	Diskettenbezogene Kommandos	33
5.4.1.	KILL	33
5.4.2.	LOAD	34
5.4.3.	MERGE	34
5.4.4.	NAME	35
5.4.5.	SAVE	35
5.4.6.	RESET	36
6.	Anweisungen	37
6.1.	Kommentaranweisung REM	37

		Seite
6.2.	Typdefinitionsanweisungen DEFINT, DEFSNG, DEFDBL, DEFSTR	37
6.3.	Wertzuweisungen	38
6.3.1.	LET	38
6.3.2.	SWAP	39
6.3.3.	MID$	39
6.3.4.	RANDOMIZE	40
6.4.	Dialog- Ein-/Ausgabe	41
6.4.1.	INPUT	41
6.4.2.	LINE INPUT	42
6.4.3.	PRINT	43
6.4.4.	PRINT USING	45
6.4.5.	LPRINT und LPRINT USING	49
6.4.6.	WRITE	50
6.4.7.	WIDTH	50
6.5.	Arbeit mit programminternen Daten	51
6.5.1.	DATA	51
6.5.2.	READ	51
6.5.3.	RESTORE	52
6.6.	Steueranweisungen	53
6.6.1.	END	53
6.6.2.	STOP	53
6.6.3.	GOTO	54
6.6.4.	GOSUB...RETURN	54
6.6.5.	ON...GOSUB und ON...GOTO	55
6.6.6.	FOR...NEXT	56
6.6.7.	WHILE...WEND	59
6.6.8.	IF...THEN[...ELSE] und IF..GOTO[...ELSE]	60
6.7.	Dimensionen von Feldern	61
6.7.1.	DIM	61
6.7.2.	OPTION BASE	62
6.7.3.	ERASE	62
6.8.	Anwendereigene Funktionsdefinition DEF FN	63
6.9.	Programmueberlagerung	64
6.9.1.	CHAIN	64
6.9.2.	COMMON	65
6.10.	Fehlerbehandlung	66
6.10.1.	ERROR	66
6.10.2.	Die Variablen ERR und ERL	66
6.10.3.	ON ERROR GOTO	66
6.10.4.	RESUME	67
7.	Standardfunktionen	69
7.1.	Allgemeines	69
7.2.	Algebraische Standardfunktionen	71
7.3.	Trigonometrische Standardfunktionen	72
7.4.	Konvertierungsfunktionen, die einen numerischen Wert liefern	73
7.5.	Konvertierungsfunktionen, die einen Zeichenkettenwert liefern	75
7.6.	Standardfunktionen fuer die Zeichenkettenverarbeitung	76
7.7.	Servicefunktionen	79
7.8.	Funktionen zur Drucker-/Bildschirmsteuerung	80
8.	Anweisungen und Funktionen fuer die Dateiarbeit	82
8.1.	Allgemeines	82
8.2.	Eroeffnen und Schliessen von Dateien	83

		Seite
8.2.1.	OPEN	83
8.2.2.	CLOSE	84
8.3.	E/A mit sequentiellen Dateien	84
8.3.1.	PRINT# und PRINT# USING	85
8.3.2.	WRITE#	86
8.3.3.	INPUT#	87
8.3.4.	LINE INPUT#	88
8.3.5.	Erstellen, Verarbeiten und Erweitern von sequentiellen Dateien	89
8.4.	E/A mit Direktzugriffsdateien	91
8.4.1.	FIELD	92
8.4.2.	LSET und RSET	92
8.4.3.	PUT	93
8.4.4.	GET	94
8.4.5.	MKI¤, MKS¤, MKD¤	94
8.4.6.	CVI, CVS, CVD	95
8.4.7.	Erstellen und Verarbeiten von Direktzugriffsdateien	96
8.5.	Funktionen	97
8.5.1.	EOF	97
8.5.2.	LOC	97
8.5.3.	LOF	98
8.5.4.	INPUT¤	98
8.5.5.	VARPTR	98
9.	Rechnerspezifische Ausdrucksmittel	100
9.1.	Codeprogramme	100
9.1.1.	CALL-Anweisung	101
9.1.2.	USR-Funktionsruf	104
9.1.3.	DEF-USR-Anweisung	106
9.1.4.	VARPTR-Funktion	106
9.2.	DEF-SEG-Anweisung	107
9.3.	PEEK-Funktion	107
9.4.	POKE-Anweisung	108
9.5.	INP-Funktion	108
9.6.	OUT-Anweisung	109
9.7.	WAIT-Anweisung	109
Anlage 1:	Fehlercodes und Fehlermeldungen	110
Anlage 2:	Syntaxzusammenstellung	114
Anlage 3:	Reservierte Woerter	120
Anlage 4:	Beispiel fuer die Arbeit mit Direktzugriffsdateien	121
Anlage 5:	Unterschiede zwischen BASIC-1700 und BASIC fuer SCPX 1520	127
Sachwortverzeichnis		128

Bildverzeichnis

			Seite
Bild	1	Kellerzustand zur Zeit der Abarbeitung der CALL-Anweisung	102
Bild	2	Kellerzustand zur Zeit der Abarbeitung des Codeprogramms	102

Tabellenverzeichnis

			Seite
Tabelle	1	Standardfunktionen	69

1. Allgemeine Informationen

1.1. Beschreibungsform der Sprache

Die Syntax der Kommandos und Anweisungen wird nach folgenden Regeln beschrieben:

- Syntaktische Einheiten in Grossbuchstaben muessen so, wie in der Syntaxbeschreibung auch im Programm erscheinen.
- Kleingeschriebene und in spitze Klammern (< >) eingeschlossene syntaktische Einheiten muss der Benutzer durch entsprechende Angaben (s. Semantik) ersetzen.
- Einheiten in eckigen Klammern ([ ]) sind wahlweise.
- Alle anderen Zeichen, mit Ausnahme spitzer und eckiger Klammern, sind zu uebernehmen (z. B. Kommas, Semikolons, Klammern, Gleichheitszeichen...).
- Syntaktische Einheiten, nach denen drei Punkte folgen (...), koennen so oft wiederholt werden, wie es die Zeilenlaenge zulaesst.
- In den Beispielen sind zur Unterscheidung zwischen Ausgaben von BASIC und Eingaben des Anwenders die Anwendereingaben unterstrichen.

Beispiel

```
RUN
? 1,4,6
Die Summe ist 11.
```

1.2. Start des Interpreters

Der Start des BASIC-Interpreters erfolgt durch die Eingabe:

```
> BASIC <CR>
```

BASIC meldet sich mit der Ausschrift:

```
BASIC-1700  V01.00
62367 Bytes free
Ok
```

Beim Start des BASIC-Interpreters koennen mehrere Optionen angegeben werden. Die Kommandozeile zum Start von BASIC laesst sich wie folgt erweitern.

```
> BASIC [<dateispezifikation>][/F:<Anzahl der Dateien>]
                              [/M:<hoechste Speicheradresse>]
                              [/S:<maximale Satzlaenge>]
```

Ist eine Programmdatei spezifiziert, so wird dieses Programm nach der Initialisierung des BASIC-Interpreters sofort eingelesen und abgearbeitet. Dabei wird genauso verfahren, als waere nach dem Aufruf von BASIC das Kommando RUN <dateispezifikation> eingegeben worden.

BASIC-Programme, die so gerufen werden, sollten mit einer SYSTEM-Anweisung, die die Rueckkehr ins Betriebssystem bewirkt, enden. Auf diese Art und Weise koennen BASIC-Programme in Stapelverarbeitung unter SUBMIT abgearbeitet werden.

<dateispezifikation>:=[g:]dateiname[.typ]

dateiname	-	maximal 8 Zeichen lang
g:	-	Geraetespezifikation, gibt das anzuwaehlende Diskettenlaufwerk (A:,B:,...E:) an
.typ	-	Dateityp (3 Zeichen lang)

Fehlt der Dateityp, so wird bei den Kommandos RUN, LOAD, SAVE und MERGE der Dateityp .BAS angenommen.
Ist kein Geraet spezifiziert, so wird mit dem gerade aktuellen Geraet gearbeitet.

Mit der F-Option wird die Anzahl der zu errichtenden Dateibeschreibungsbloecke festgelegt. Es muessen so viele Dateibeschreibungsbloecke angelegt werden, dass fuer jede zu eroeffnende Datei ein Dateibeschreibungsblock zur Verfuegung steht. Beim Schliessen einer Datei wird deren Dateibeschreibungsblock freigegeben und kann fuer weitere Dateien verwendet werden. Jeder Dateibeschreibungsblock erfordert 166 Byte Speicherplatz. Ist keine F-Option angegeben, so werden drei Dateibeschreibungsbloecke angelegt (s. Abschn. 8.2.1.).

Durch die M-Option wird die hoechste vom BASIC-System nutzbare Speicheradresse festgelegt. Fehlt die M-Option, so steht der gesamte freie Speicher zur Verfuegung.

Mit Hilfe der S-Option wird die maximale Satzlaenge fuer Direktzugriffsdateien eingestellt. Standardmaessig betraegt diese Satzlaenge 128 Byte.

Hinweis

Die Werte fuer <Anzahl der Dateien>
<hoechste Speicheradresse> und
<maximale Satzlaenge>

koennen dezimal, oktal (mit vorangestelltem &0) oder hexadezimal (mit vorangestelltem &H) angegeben werden.

Beispiele

>BASIC TEST.BAS	Der gesammte freie Speicher und 3 Dateien stehen zur Verfuegung. TEST.BAS wird geladen und ausgefuehrt.
>BASIC BEISPIEL/F:6/M:&H8000	Die ersten 32K des Speichers und 6 Dateibeschreibungsbloecke stehen zur Verfuegung. Das Programm BEISPIEL.BAS wird geladen und ausgefuehrt.

1.3. Operationsmodi

Nach dem Start von BASIC-1700 wird die Anforderung "Ok" ausgeschrieben. BASIC-1700 befindet sich dann auf dem Kommandoniveau, d.h. es werden Eingaben erwartet. BASIC-1700 kann nun entweder im direkten oder im indirekten Modus verwendet werden.
Im direkten Modus werden den BASIC-Anweisungen und -Kommandos keine Zeilennummerm vorangestellt. Sie werden sofort nach der Eingabe ausgefuehrt. Resultate von arithmetischen und logischen Operationen koennen unmittelbar angezeigt werden und fuer eine spaetere Verwendung gespeichert werden. Der Text der Instruktion wird nicht aufbewahrt. Er ist nach der Ausfuehrung nicht mehr vorhanden. Dieser Modus ist zum Testen von BASIC-Programmen und zur Verwendung von BASIC als "Taschenrechner" fuer schnelle Berechnungen sinnvoll.
Der indirekte Modus wird zur Eingabe von Programmen verwendet. Den Programmzeilen wird eine Zeilennummer vorangestellt, und sie werden im Speicher abgelegt. Das im Speicher abgelegte Programm wird nach Eingabe eines RUN-Kommandos abgearbeitet.

1.4. Eingabe von Programmen

BASIC-Programme koennen ueber das Bediengeraet oder von einer Diskettendatei (siehe diskettenbezogene Kommandos) eingegeben werden. Bei der Eingabe sind die im Anschluss beschriebenen syntaktischen Vorschriften fuer BASIC-Programme zu beruecksichtigen. Jede Programmzeile muss mit einer Zeilennummer beginnen und ist mit einem "carriage return" abzuschliessen. Eine Programmzeile kann aus maximal 255 Zeichen bestehen.
Die Programmzeilen koennen in beliebiger Reihenfolge eingegeben werden. Entsprechend ihrer Zeilennummer werden sie bei der Eingabe in den Arbeitsspeicher sortiert. Wird eine Zeilennummer erneut verwendet, so wird die alte Zeile durch die eingegebene neue Zeile ersetzt. Auf diese Weise koennen fehlerhafte Programmzeilen eines sich im Arbeitsspeicher befindlichen Programms durch einfaches Ueberschreiben ersetzt (korrigiert) werden.
Wird ein unkorrektes Zeichen eingegeben, waehrend die Zeile noch geschrieben wird, so kann es mit der RUBOUT-Taste oder Control-H geloescht werden. RUBOUT schliesst die geloeschten Zeichen mit Ruckwaertsschraegstrichen (\) ein. Control-H fuehrt einen Rueckschritt aus und loescht das Zeichen. Sobald das Zeichen (ggf. mehrere) geloescht ist, kann das Schreiben der Zeile fortgesetzt werden. Zum Loeschen einer Zeile, die noch geschrieben wird, muss Control-U eingegeben werden. Ein <carriage return> wird automatisch ausgefuehrt, sobald die Zeile geloescht ist. (Weitere Editierungsmoeglichkeiten siehe EDIT, Abschnitt 5.2.3.).
Zum Loeschen des vollstaendigen Programms, das sich gerade im Speicher befindet, wird das NEW-Kommando verwendet (s. Abschn. 5.2.7.). NEW wird benutzt, um vor Eingabe eines neuen Programms den Speicher zu loeschen.

1.5. Fehlermeldungen

Wenn BASIC-1700 einen Fehler erkennt, der zum Abbruch der Programmabarbeitung fuehrt, wird eine vollstaendige Fehlermeldung ausgegeben. Eine komplette Liste der Fehlercodes und Fehlermeldungen ist in Anlage 1 enthalten.

2. Programmstruktur

2.1. Zeilenformat

Ein BASIC-Programm besteht aus einer Folge von Programmzeilen. Die Programmzeilen haben das folgende Format:

nnnn BASIC-Anweisung [:BASIC-Anweisung...]<carriage return>

Auf einer Zeile koennen mehrere BASIC-Anweisungen angegeben werden, jedoch muss jede Anweisung von der vorangehenden durch einen Doppelpunkt getrennt werden.
Eine BASIC-Programmzeile beginnt immer mit der Zeilennummer, endet mit einem <carriage return> und kann maximal 255 Zeichen enthalten. Es ist moeglich, eine logische Zeile ueber mehr als eine physische Zeile fortzufuehren. Dazu wird das Zeichen <line feed> benutzt. Dieses Zeichen erlaubt die Fortsetzung der Eingabe einer logischen Zeile auf der naechsten physischen Zeile.

2.2. Zeilennummern

Jede BASIC-Programmzeile beginnt mit einer Zeilennummer. Die Zeilennummern legen die Reihenfolge fest, in der die Programmzeilen im Speicher gespeichert werden. Ueber die Zeilennummer erfolgt der Zugriff zur Zeile bei den Kommandos zur Editierung, zum Auslisten von Programmteilen usw. Die Zeilennummmer markiert die Zeile. In Verzweigungen wird auf die Zeilennummer Bezug genommen. Zeilennummern muessen im Bereich von 0 bis 65529 liegen. Der Punkt (.) kann in den Kommandos EDIT und LIST anstelle einer Zeilennummer zum Referieren der aktuellen Zeile verwendet werden.

3. Grundelemente

3.1. Zeichensatz

Der BASIC-1700-Zeichensatz umfasst alphabetische Zeichen, Ziffern und Spezialzeichen.
Die alphabetischen Zeichen in BASIC-1700 sind die Klein- und Grossbuchstaben des Alphabets.
Die folgenden Spezialzeichen und Terminaltasten werden von BASIC-1700 akzeptiert:

Zeichen	Name bzw. Verwendung
	Leerzeichen
=	gleich bzw. Zuweisungszeichen
+	plus
-	minus
*	Stern bzw. Multiplikationssymbol
/	Schraegstrich bzw. Divisionssymbol
^	Pfeil bzw. Potenzierungssymbol
(	oeffnende runde Klammer
)	schliessende runde Klammer
%	Prozent
#	Doppelkreuz
¤	Dollarsymbol
!	Ausrufezeichen
[	oeffnende eckige Klammer
]	schliessende eckige Klammer
,	Komma
.	Punkt bzw. Dezimalpunkt
'	Apostroph
"	Anfuehrungsstriche
;	Semikolon
:	Doppelpunkt
&	Ampersand
?	Fragezeichen
<	kleiner
>	groesser
\	Rueckwaertsschraegstrich bzw. Symbol fuer ganzzahlige Division
@	kaufmaennisches A
_	Unterstreichung
<rubout>	loescht das letzte geschriebene Zeichen
<escape>	fuer Kommandos im Escape Edit Modus
<tab>	bewegt die Druckposition zur naechsten Tabulatorposition
<line feed>	Uebergang auf die naechste physische Zeile
<carriage return>	Ende der Eingabe einer Zeile

3.2. Steuerzeichen

Control-A	Eintritt in den Editermodus fuer die gerade geschriebene Zeile oder Dateneingabe
Control-C	Unterbrechung der Programmausfuehrung und Rueckkehr zum BASIC-1700-Kommandoniveau
Control-H	Rueckwaertsschritt; loeschen des letzten geschriebenen Zeichen
Control-I	Tabulator; Tabulatorpositionen sind nach jeweils 8 Zeichen angeordnet
Control-R	Wiederholung der Zeile, die gerade geschrieben wurde
Control-S	Unterbrechung der Programmausfuehrung
Control-Q	Fortsetzung der Programmausfuehrung nach einem Control-S
Control-U	Loeschen der Zeile, die gerade geschrieben wurde

3.3. Konstanten

Konstanten sind rechnerexterne Darstellungen von Werten.
BASIC-1700 unterscheidet:
- Numerische Konstanten und
- Zeichenkettenkonstanten (String)

Zeichenkettenkonstanten (String-Konstanten) sind Folgen von maximal 255 alphanumerischen Zeichen, die in Anfuehrungsstriche (") eingeschlossen sind.

Beispiele

```
"A STRING CONSTANT"
"0123456789ABCDEF"
```

Innerhalb einer Zeichenkettenkonstanten kann ein Anfuehrungsstrich (") selbst nicht auftreten.
Zwei aufeinanderfolgende Anfuehrungsstriche bilden eine leere String-Konstante (String-Konstante mit der Laenge 0).
Falls im Anschluss an eine Zeichenkettenkonstante die Programmzeile beendet wird, kann der abschliessende Anfuehrungsstrich entfallen.

Numerische Konstanten sind positive oder negative Zahlen.
Numerische Konstanten duerfen in BASIC-1700 kein Komma enthalten.
Es gibt fuenf Typen von numerischen Konstanten:
- Ganzzahlige Konstanten (Integer-Konstanten)
  Ganze Zahlen zwischen -32767 und +32767. Integer-Konstanten duerfen keinen Dezimalpunkt enthalten.

- Reelle Konstanten in Festpunktdarstellung
  Positive oder negative reelle Zahlen, d. h. Zahlen, die einen Dezimalpunkt enthalten.
- Reelle Konstanten in halblogarithmischer Darstellung (Gleitpunktkonstanten)
  Positive oder negative Zahlen in halblogarithmischer Darstellung. Eine Gleitpunktkonstante besteht aus einer wahlweise vorzeichenbehafteten Integer- oder reellen Konstanten in Festpunktdarstellung (Mantisse), gefolgt vom Buchstaben E und einer wahlweise vorzeichenbehafteten Integer-Konstanten (Exponent).
  Der moegliche Bereich fuer diese Konstanten ist 10^{-38} bis 10^{+38}.

  Beispiele

  651.243E-7 = 0.000651243
  9357E6 = 9357000000
  (Gleitpunktkonstanten doppelter Genauigkeit verwenden den Buchstaben D anstelle von E.

- Hexadezimalkonstanten
  Hexadezimalzahlen mit vorangehenden &H.

  Beispiele

  &H67
  &HF23

- Oktalkonstanten
  Oktalzahlen mit vorangehenden &O bzw. &.

  Beispiele

  &0734
  &1001

Numerische Konstanten koennen rechnerintern entweder mit einfacher oder mit doppelter Genauigkeit gespeichert werden.
Bei einfacher Genauigkeit werden 7 Stellen abgebildet. Die Genauigkeitsangabe von 7 oder 16 Stellen bezieht sich ausschliesslich darauf, dass eingegebene und intern naeherungsweise dargestellte Zahlen bei der Ausgabe bis zur 7. oder 16. Stelle originalgetreu reproduziert werden.
Bei Berechnungen koennen sich Genauigkeitsverluste auf Grund der naeherungsweisen Darstellung auch auf Stellen vor der 7. bzw. 16. Stelle auswirken.

Beispiel

```
10 FOR I=1 TO 50000
20 S=S+0.01
30 NEXT I
40 PRINT S
RUN
---
 500.172
```

Eine einfach genaue Konstante ist eine numerische Konstante, fuer

die eine der folgenden Bedingungen zutrifft:

- Sieben oder weniger Ziffern
- Halblogarithmische Darstellung mit E
- Ein nachfolgendes Ausrufezeichen (!)

Eine doppelt genaue Konstante ist eine numerische Konstante, fuer die eine der folgenden Bedingungen zutrifft:

- Acht oder mehr Ziffern
- Halblogarithmische Darstellung mit D
- Ein nachfolgendes Doppelkreuz (#)

Beispiele

einfach genaue Konstanten	doppelt genaue Konstanten
53.9	987643210
-1.11E-03	-1.11468D-02
1256.0	1256.0#
25.8!	1234567.9876

3.4. Variable

Variablen sind Groessen, die durch einen Namen identifiziert werden und die einen Wert besitzen. Dieser Wert kann der Variablen durch eine Wertzuweisung oder eine Eingabeanweisung zugewiesen werden. Bevor einer Variablen ein Wert zugewiesen wurde, ist ihr Wert 0 bei numerischen Variablen, bzw. eine leere Zeichenreihe bei String-Variablen.

3.4.1. Variablennamen und Deklarationszeichen

Variablennamen koennen eine beliebige Laenge haben, jedoch sind nur 40 Zeichen signifikant. Ein Variablenname kann aus Buchstaben, Ziffern und dem Zeichen "." bestehen. Das erste Zeichen muss ein Buchstabe sein. Das letzte Zeichen kann ein spezielles Typdeklarationszeichen sein.
Variablennamen duerfen nicht mit reservierten Woertern identisch sein und nicht mit FN beginnen. Beginnt ein Name mit FN, wird das als Ruf einer nutzerdefinierten Funktion aufgefasst. Reservierte Woerter sind alle BASIC-Kommando-, -Anweisungs-, -Funktions- und -Operatornamen . Reservierte Woerter muessen durch Leerzeichen von den Variablennamen getrennt werden. Nach den Wertarten in BASIC-1700 werden numerische Variablen (Integer-Variablen, Variablen einfacher oder doppelter Genauigkeit) und String-Variablen unterschieden. String-Variablennamen muessen als letztes Zeichen ein Dollarzeichen (¤) aufweisen, z.B. A¤. Das Dollarzeichen ist ein Typdeklarationszeichen, d.h. es deklariert die Variable als Variable deren Werte Zeichenketten sind. Fuer numerische Variablen werden folgende Typdeklarationszeichen verwendet:

- % fuer Integer-Variable
- ! fuer Variable einfacher Genauigkeit
- # fuer Variable doppelter Genauigkeit

Ist kein Typdeklarationszeichen angegeben, so wird einfache Genauigkeit angenommen.

Beispiele

PI#	deklariert einen doppelt genauen Wert
Bound.of.area!	deklariert einen einfach genauen Wert
median%	deklariert einen Integer-Wert
NAME¤	deklariert einen String-Wert
EPS	deklariert einen einfach genauen Wert

Es gibt eine zweite Methode zur Deklaration von Variablentypen. Die Anweisungen DEFINT, DEFSTR, DEFSNG und DEFDBL koennen innerhalb eines Programms zur Deklaration von Typen fuer verschiedene Variablennamen verwendet werden. Beschreibung dieser Anweisung (s. Abschn. 6.12.).

3.4.2. Feldvariable

Felder sind geordnete Folgen von Werten, die durch den gleichen Variablennamen referiert werden. Jedes Element eines Feldes wird durch eine Feldvariable referiert, die durch Integer-Werte oder Integer-Ausdruecke indiziert ist. Ein Feldvariablenname hat soviele Indizes wie Dimensionen im Feld vorhanden sind. Zum Beispiel referiert V(10) einen Wert in einem eindimensionalen Feld, T(1,4) einen Wert in einem zweidimensionalen Feld usw. Ein Feld kann im Rahmen des verfuegbaren Speicherplatzes beliebig viele Dimensionen und Elemente in einer Dimension besitzen.

3.4.3. Speicherplatzbedarf

Variablenart:	Speicherplatzbedarf in Byte:
- Integer	2
- Einfache Genauigkeit	4
- Doppelte Genauigkeit	8
Felder:	
- Integer	2 pro Element
- Einfache Genauigkeit	4 pro Element
- Doppelte Genauigkeit	8 pro Element
Zeichenketten:	3 Byte plus aktuelle Laenge der Zeichenkette

Hinweis

Feldvariablen werden im Speicher im Anschluss an die einfachen Variablen gefuehrt. Durch das Einfuegen von einfachen Variablen veraendern sich die Adressen bereits definierter Felder. Dies ist zu beachten, wenn beim Aufrufen von Codeunterprogrammen (s. Abschn. 9.) Feldvariablen als Parameter uebergeben werden sollen. Alle im Programm auftretenden einfachen Variablen muessen vor dem Unterprogrammaufruf bzw. vor einer auf ein Feld angewand-

ten VARPTR-Funktion verwendet worden sein.

3.5. Typ-Konvertierungen

BASIC-1700 konvertiert bei Bedarf numerische Werte von einem Typ zu einem anderen. Dabei gelten folgende Regeln:

- Wenn ein numerischer Wert eines Typs einer numerischen Variablen eines anderen Typs zugewiesen werden soll, so wird dieser Wert in den Typ der Variablen konvertiert (Die Zuweisung eines numerischen Wertes an eine String-Variable bzw. umgekehrt fuehrt zum Fehler "Type mismatch").
- Bei der Konvertierung eines Gleitpunktwertes in einen Integer-Wert wird der gebrochene Anteil gerundet.

Beispiele

```
10 A1%=-1.8 : A2%=-1.5 : A3%=-1.2
20 B1%=+1.8 : B2%=+1.5 : B3%=+1.2
30 PRINT A1%;A2%;A3%
40 PRINT B1%;B2%;B3%

RUN
 -2 -2 -1
  2  2  1
```

- Wenn einer doppelt genauen Variablen ein einfach genauer Wert zugewiesen wird, so sind nur die ersten sieben Stellen des konvertierten Wertes nach einer Rundung gueltig. Dies folgt aus der Tatsache, dass ein einfacher Wert nur 7 Stellen genau ist. Die externe Darstellung des einfach genauen Wertes ist mit der des daraus gewonnenen doppelt genauen Wertes meist nicht identisch. Im Gleitpunktformat werden die dezimal notierten Zahlen intern nur naeherungsweise binaer dargestellt. Die Differenz zwischen notiertem Wert und interner Darstellung wird durch den sogenannten Rundungsausgleich bei der Ausgabekonvertierung in der Regel ausgeglichen, d.h. es erscheint die Zahl so, wie sie urspruenglich notiert wurde. Der Rundungsausgleich wirkt bei einfacher Genauigkeit auf die 7. Stelle, bei doppelter Genauigkeit jedoch erst auf die 16. Stelle. Dadurch werden bei der Konvertierung einfach genauer Zahlen in das doppelt genaue Format Darstellungsungenauigkeiten sichtbar.
- Waehrend der Auswertung von Ausdruecken werden alle Operanden arithmetischer und Vergleichsausdruecke auf den gleichen Genauigkeitsgrad konvertiert, d.h. auf den des genauesten Operanden. Das Resultat eines arithmetischen Ausdruckes liegt in diesem Genauigkeitsgrad vor.

Beispiele

```
10 Q#=23#/19      Die Berechnung wird in doppelter Genauig-
20 PRINT Q#       keit ausgefuehrt und das Resultat in
                  doppelter Genauigkeit in Q# abgespeichert.
RUN
1.210526315789474
```

```
10 Q=23#/19
20 PRINT Q

RUN
---
1.210526
```

Die Berechnung wird in doppelter Genauigkeit ausgefuehrt, das Resultat zu einfacher Genauigkeit gerundet und anschliessend der einfach genauen Variable Q zugewiesen.

- Logische Werte werden bei Bedarf zu numerischen Werten konvertiert und umgekehrt. Diese Konvertierungen sind im Abschnitt 4.4. beschrieben.

4. Ausdruecke und Operatoren

4.1. Allgemeines

Ausdruecke sind die programmiersprachliche Notation mathematischer Formeln. Jeder Ausdruck liefert bei seiner Abarbeitung einen Wert. Nach der Art des gelieferten Wertes werden folgende Ausdruecke unterschieden:

- numerische Ausdruecke (der Wert ist vom Typ Integer oder Real einfacher oder doppelter Genauigkeit)
- String-Ausdruecke (der berechnete Wert ist eine Zeichenkette)
- logische Ausdruecke (der berechnete Wert ist ein logischer Wert)

Jeder Ausdruck besteht aus Operanden und Operatoren. Die Wertart der Operanden und die verwendeten Operatoren bestimmen die Art des Ausdruckes. Die vorhandenen Operatoren koennen in fuenf Klassen eingeteilt werden:

- arithmetische Operatoren
- Zeichenkettenoperator
- Vergleichsoperatoren
- logische Operatoren
- funktionale Operatoren

4.2. Arithmetische Operatoren

Es gibt folgende arithmetische Operatoren (geordnet nach ihrer Prioritaet):

Operator	Operation	Beispiel
^	Potenzierung	X^Y
-	Vorzeichenumkehr	-X
* /	Multiplikation, Gleitpunktdivision	X*Y X/Y
\	ganzzahlige Division	X\Y
MOD	Modulo	X MOD Y
+ -	Addition Subtraktion	X+Y X-Y

Die Reihenfolge der Ausfuehrung der Operationen kann durch runde Klammern beeinflusst werden. Innerhalb der Klammern wird die uebliche Reihenfolge beibehalten.

Beispiele
(Algebraische Ausdruecke und ihre BASIC-Entsprechung)

Algebraischer Ausdruck	BASIC-Ausdruck
X+2Y	X+2*Y
$X-\frac{Y}{2}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z-Y}$	(X+Y)/(Z-Y)
$(X^2)^Y$	(X*X)^Y
X^{Y^Z}	X^(Y^Z)
X(-Y)	X*(-Y) (Zwei aufeinander folgende Operatoren muessen durch Klammern getrennt werden).

Die Integer-Division wird durch den Rueckwaertsschreagstrich (\) ausgedrueckt. Die Operanden werden zu Integer-Werten gerundet und muessen im Bereich von -32768 bis 32767 liegen. Anschliessend wird die Division ausgefuehrt, wobei das Resultat sich aus dem ganzzahligen Anteil des Quotienten ergibt.

Beispiele

10\4=2
25.68\6.99=3

Die Prioritaet der Integer-Division liegt unmittelbar nach der Prioritaet von Multiplikation und Gleitpunktdivision.

Die Berechnung des Divisionsrestes erfolgt durch den Operator MOD. Er liefert einen Integer-Wert, der der Rest einer Integer-Division ist.

Beispiele

114.6 MOD 7=3 (115/7=16 mit Rest 3)
9.5 MOD 4.1=2 (10/4=2 mit Rest 2)

Die Prioritaet des MOD-Operators liegt unmittelbar nach der Prioritaet der Integer-Division.

Ueberlauf und Division durch Null

Tritt waehrend der Berechnung eines Ausdruckes eine Division durch Null auf, so wird der Fehler "Division by Zero" angezeigt. Die groesste darstellbare Zahl mit dem Vorzeichen des Dividenden wird als Resultat der Division angenommen und die Berechnung fortgesetzt. Wenn bei der Berechnung einer Potenzierung eine Basis Null und ein negativer Exponent auftritt, wird ebenfalls der Fehler "Division by Zero" angezeigt. Die groesste darstellbare positive Zahl wird als Resultat der Potenzierung angenommen und die Berechnung fortgesetzt.
Tritt ein Ueberlauf ein, so wird der Fehler "Overflow" angezeigt. Die groesste darstellbare Zahl mit dem algebraisch korrekten Vorzeichen wird als Resultat angenommen und die Berechnung fortgesetzt. Beide Fehler koennen nicht durch eigene Fehlerbehandlungsroutinen bearbeitet werden.

4.3. Vergleichsoperatoren

Vergleichsoperatoren werden zum Vergleich zweier Werte verwendet. Das Resultat eines Vergleichs ist entweder der logische Wert "wahr" oder der logische Wert "falsch". Es kann benutzt werden, um eine Entscheidung ueber den Programmablauf zu treffen (s. Abschn. 6.7.).

Operator	Vergleichsbedingung	Ausdruck
=	gleich	X=Y
<> oder ><	ungleich	X<>Y X><Y
<	kleiner als	X<Y
>	groesser als	X>Y
<= oder +<	kleiner oder gleich	X<=Y, X=<Y
>= oder =>	groesser oder gleich	X>=Y, X=>Y

Das Gleichheitszeichen wird mit anderer Bedeutung auch zur Zuweisung eines Wertes an eine Variable verwendet (s. Abschn. 6.3.1.).

Wenn arithmetische und Vergleichsoperatoren in einem Ausdruck kombiniert werden, so werden die arithmetischen stets zuerst ausgefuehrt, d.h. die Vergleichsoperatoren haben eine niedrigere Prioritaet als die numerischen Operatoren "+" und "-". Zum Beispiel ist der Ausdruck

```
X+Y < (T-1)/Z
```

wahr, wenn der Wert von X plus Y kleiner ist als der Wert von T-1 dividiert durch Z.

Beispiele

```
IF SIN(X) < 0 GOTO 1000
IF I MOD J <> 0 THEN K=K+1
```

4.4. Logische Operatoren

Logische Operatoren koennen sowohl zur Verknuepfung aus Vergleichsausdruecken gewonnener logischer Werte als auch zur bitweisen Manipulation von Integer-Werten verwendet werden. Dabei wird die Moeglichkeit ausgenutzt, numerische Werte des Typs Integer als logische Werte und umgekehrt, logische Werte als Integer-Werte zu betrachten. Ein Integer-Wert ungleich Null wird als logischer Wert "wahr", der Interger-Wert Null als logischer Wert "falsch" gedeutet. Treten im Zusammenhang mit logischen Operanden numerische Werte vom Typ Real auf, so werden diese vorher zu Integer konvertiert. Diese Konvertierung ist nur moeglich, wenn der Real-Wert im Bereich von -32768 bis +32767 liegt. Werden logische Operatoren auf Real-Werte angewandt, die ausserhalb dieses Bereiches liegen, tritt ein Fehler auf. Wird bei Vergleichen der logische Wert "wahr" ermittelt, so entspricht das dem numerischen Wert -1 (alle 16 bit der internen Darstellung gesetzt). Wird bei einem Vergleich das Ergebnis "falsch" berechnet, so entspricht das dem numerischen Wert 0 (alle bit der internen Darstellung geloescht). Die logischen Operatoren wirken bitweise auf die 16 bit des oder der logischen bzw. Integer-Operanden. In einem Ausdruck werden logische Operationen nach arithmetischen und Vergleichsoperationen ausgefuehrt. Das Ergebnis einer logischen Operation wird in den folgenden Tabellen beschrieben. Die Operationen sind nach Prioritaeten geordnet.

NOT

X	NOT X
1	0
0	1

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

IMP

X	Y	X IMP Y
1	1	1
1	0	0
0	1	1
0	0	1

EQV

X	Y	X EQV Y
1	1	1
1	0	0
0	1	0
0	0	1

Beispiele

```
IF D < 200 AND F < 4 THEN 80
IF I > 100 OR K < 0 THEN 50
IF NOT P THEN 100
```

Logische Operatoren konvertieren ihre Operanden zu vorzeichenbehafteten Integer-Werten von 16 Bit Laenge im Bereich von -32768 bis +32767. Liegen die Operanden nicht in diesem Bereich, so wird ein Fehler angezeigt. Wenn beide Operanden die Werte 0 oder -1 besitzen, liefern die logischen Operatoren ebenfalls 0 oder -1. Die Operationen werden ueber den Integer-Werten in bitweiser Form ausgefuehrt, d.h. jedes Bit des Resultates wird aus den korrespondierenden Bits der beiden Operanden bestimmt.
Deshalb ist es moeglich, logische Operatoren zum Bytetest fuer einzelne Bitmuster zu verwenden. Zum Beispiel kann der AND-Operator zum Maskieren der Bits des Statusbyte eines Maschinen-IO-Portes verwendet werden. Der OR-Operator kann zum "Mischen" zweier Byte zur Erstellung eines speziellen Binaerwertes verwendet werden.

Die folgenden Beispiele demonstrieren die Wirkungsweise der logischen Operatoren.

```
13 AND 10 = 8       13 = binaer 1101, 10 = binaer 1010
                    das binaere Ergebnis ist 1000, dezimal 8

5 AND 9 = 1         5 = binaer 101, 9 = binaer 1001
                    das Ergebnis ist 1

13 OR 10 =15        13 = binaer 1101, 10 = binaer 1010
                    das binaere Ergebnis ist 1111, dezimal 15

5 XOR 9 = 12        5 =binaer 0101, 9 = binaer 1001
                    das binaere Ergebnis ist 1101, dezimal 13

6 IMP 24 = -7        6 = binaer 0000000000000110
                    24 = binaer 0000000000011000
                    Ergebnis   1111111111111001, dezimal -7

NOT 6 = -7          (Binaerdarstellung s. voriges Beispiel)
```

4.5. Funktionale Operatoren

Ein Ausdruck kann Funktionsrufe zum Aktivieren vordefinierter Operationsfolgen, die ueber die Operanden ausgefuehrt werden sollen, enthalten.
BASIC-1700 besitzt Standardfunktionen, die im System vorhanden sind (s. Abschn. 7.). Weiterhin erlaubt BASIC-1700 Anwenderfunktionen, die vom Programmierer geschrieben werden koennen (siehe DEF FN).

4.6. String-Ausdruecke und String-Vergleich

String-Werte koennen mit dem Operator (+) verkettet werden.

Beispiel

```
10 FILENAME¤="MYFILE" : FILETYPE¤=".BAS"
20 PRINT FILENAME¤+FILETYPE¤
30 DEVICE¤="A:"
40 PRINT DEVICE¤+FILENAME¤+FILETYPE¤

RUN
MYFILE.BAS
A:MYFILE.BAS
```

String-Werte koennen mit den gleichen Vergleichsoperatoren verglichen werden wie numerische Werte:

```
=   <>   <   >   <=   >=
```

String-Vergleiche erfolgen zeichenweise von links nach rechts bis:
- das Ende einer der beiden Zeichenketten erreicht wird oder
- zwei unterschiedliche Zeichen gefunden werden.

Wird der Vergleich am Ende einer der beiden Zeichenketten beendet, so ist das Vergleichsergebnis "gleich", wenn beide Zeichenketten die gleiche Laenge haben, sonst wird die kuerzere Zeichenkette als die kleinere betrachtet.
Wird der Vergleich durch unterschiedliche Zeichen beendet, ist die Zeichenkette kleiner (groesser), die das kleinere (groessere) Zeichen enthaelt. Die Ordnung der Zeichen ist durch die ASCII-Codetabelle (KOI-7-Code, ST RGW 05-15) definiert.

Beispiele

```
"STAMP" < "STEAMER"
"STEAMER" < "Stamp"
"PC" = "PC"
"PC  " > "PC"
"letter Y" > "letter B"
```

String-Vergleiche koennen zum Testen von String-Werten oder zum alphabetischen Ordnen verwendet werden.
Alle String-Konstanten, die in Vergleichsausdruecken verwendet werden, muessen in Anfuehrungsstriche (") eingeschlossen sein.

5. Kommandos

5.1. Allgemeines

Alle Kommandos und Anweisungen sind nach folgendem Schema beschrieben:

Syntax	zeigt den korrekten Aufbau der Anweisung.
Semantik	beschreibt, wie die Anweisung zu benutzen ist.
Beispiel	demonstriert die Benutzung der beschriebenen Anweisung an einfachen Beispielen.

In der Regel werden Kommandos im direkten Operationsmodus des BASIC-Systems (d.h. ohne Zeilennummern) verwendet.
Es ist aber auch moeglich, und in einigen Faellen sinnvoll, Kommandos mit Zeilennummern zu versehen und ins Programm einzufuegen. Als Beispiel dafuer seien NAME, KILL und SYSTEM genannt. Nach Ausfuehrung bestimmter Kommandos kehrt BASIC in den Kommandomodus zurueck. Diese Kommandos sollten nicht in ein Programm eingefuegt werden. In den Beschreibungen der einzelnen Kommandos wird darauf hingewiesen.

5.2. Kommandos zur Programmaufbereitung

5.2.1. AUTO

Syntax

AUTO [<zeilennummer>[,[<schrittweite>]]]

Semantik

Bei Programmeingabe wird automatisch die Zeilennummer bereitgestellt. Die Numerierung beginnt bei <zeilennummer>. Jede folgende Zeilennummer wird aus der vorhergehenden durch Addition der <schrittweite> errechnet. Standard fuer <zeilennummer> und <schrittweite> ist 10. Steht nach <zeilennummer> ein Komma und fehlt die Schrittweite, so wird die letzte in einem AUTO-Kommando verwendete Schrittweite angenommen.
Wird eine Zeilennummer erzeugt, zu der schon eine Zeile existiert, so wird als Warnung ein "*" nach der Zeilennummer ausgegeben. Erfolgt trotzdem eine Eingabe, wird die vorhandene Zeile ueberschrieben. Bei Eingabe von <CR> bleibt die Zeile erhalten und es erscheint die naechste Zeilennummer.
AUTO wird durch <CTRL/C> beendet. Die Zeile, die das <CTRL/C> enthaelt, wird nicht abgespeichert. BASIC kehrt nach <CTRL/C> zum Kommandoniveau zurueck.

Beispiel

```
AUTO 910,5      erzeugt Zeilennummern 910, 915, 920...
AUTO            erzeugt Zeilennummern 10, 20, 30, 40...
```

5.2.2. DELETE

Syntax

DELETE [<zeilennummer>][-<zeilennummer>]

Semantik

Das DELETE-Kommando dient dem Loeschen einzelner oder mehrerer Zeilen aus dem aktuellen BASIC-Programm.
Bei der Angabe DELETE <zeilennummer> wird nur die durch die Zeilennummer bezeichnete Zeile geloescht. Wird DELETE <zeilennummer> -<zeilennummer> notiert, werden die Zeilen ab erster angegebener Zeilennummer bis einschliesslich zweiter angegebener Zeilennummer geloescht. DELETE -<zeilennummer> bewirkt das Loeschen ab Programmbeginn bis zur eingegebenen Zeile. DELETE ist ohne Zeilennummernangabe nicht zulaessig. Existiert zu einer Zeilennummernangabe keine entsprechende Zeile erfolgt eine Fehlermeldung. Nach der Ausfuehrung von DELETE befindet sich BASIC immer im Kommandomodus.

Beispiel

```
DELETE 120       Loeschen der Zeile 120
DELETE -300      Loeschen von Programmbeginn bis Zeile 300
DELETE 120-300   Loeschen der Zeilen von 120 bis 300
```

5.2.3. EDIT

Syntax

EDIT <zeilennummer>

oder

EDIT .

Semantik

Mit dem EDIT-Kommando koennen BASIC-Zeilen korrigiert werden, ohne dass die ganze Zeile neu eingegeben werden muss.
Das EDIT-Kommando mit Zeilennummernangabe bezieht sich auf die entsprechende Zeile. Steht anstelle der Zeilennummer das Zeichen "." bezieht sich das EDIT-Kommando auf die aktuelle Zeile. Die aktuelle Zeile ist die zuletzt eingegebene oder editierte bzw. die Zeile, bei der ein Programmlauf wegen eines Fehlers oder nach einer STOP-Anweisung unterbrochen wurde.
BASIC reagiert auf das EDIT-Kommando mit der Ausgabe der Zeilennummer der zu editierenden Zeile und geht in den EDIT-Modus ueber. Im EDIT-Modus wird die entsprechende Zeile mit EDIT-Subkommandos aufbereitet.

EDIT-Subkommandos

Es gibt EDIT-Subkommandos zum Positionieren des Kursors, zum Einfuegen von Zeilen, zum Loeschen, Ersetzen oder Aufsuchen von Text in der Zeile. Die Subkommandos erscheinen nicht auf dem Terminal. Vor den meisten der EDIT-Subkommandos kann eine ganze Zahl geschrieben werden, worauf das Subkommando entsprechend oft wiederholt wird. Fehlt diese Zahl wird das Subkommando einmal ausgefuehrt.
In der nachfolgenden Beschreibung der EDIT-Subkommandos steht:

[i] fuer den wahlweise angebbaren Wiederholungsfaktor,
<SP> fuer ein Leerzeichen (Space),
<ch> fuer ein beliebiges Zeichen,
<text> fuer eine Zeichenkette,
<ESC> fuer das ESCAPE-Zeichen und
<BSP> fuer das BACKSPACE-Zeichen.

- Kursorpositionierung

  <SP> [i]<SP> setzt den Kursor um i Zeichen nach rechts. Die Zeichen, ueber die sich der Kursor bewegt, werden dabei ausgegeben.

  <BSP> [i]<BSP> setzt den Kursor um i Zeichen nach links. Die Zeichen, ueber die sich der Kursor bewegt, werden ausgegeben.

- Einfuegen von Text

  I I<text> fuegt den <text> nach der aktuellen Kursorposition ein. Die eingefuegten Zeichen werden auf dem Terminal angezeigt. Das Einfuegen endet mit der Eingabe des ESCAPE-Zeichens. Das Zeichen <CR> beendet ebenfalls das Einfuegen und fuehrt zum Verlassen des EDIT-Modus. Zeichen, durch die die Zeile laenger als 255 Zeichen wuerde, werden abgewiesen. Mit den Zeichen <BACKSPACE>, <delete> und dem Unterstreichungszeichen koennen waehrend der Texteingabe links vom Kursor stehende Zeichen geloescht werden. Bei <BACKSPACE> werden die geloeschten Zeichen angezeigt. Bei Loeschung mit <delete> oder Unterstreichungszeichen wird das Unterstreichungszeichen ausgegeben.

  X Mit dem X-Kommando koennen Zeichen an die Zeile angehaengt werden.
  Der Kursor wird durch "X" auf das Zeilenende gesetzt. Danach kann die Zeile mit Texteingabe (siehe "I") erweitert werden. Die Beendigung der Texteingabe erfolgt durch <CR> oder <ESC>.

- Loeschen von Text

  D [i]D loescht i rechts vom Kursor stehende Zeichen. Die geloeschten Zeichen werden mit "\" begrenzt angezeigt. Der Kursor steht danach rechts vom letzten geloeschten Zeichen. Stehen rechts vom Kursor weniger als i Zeichen, so wird der Rest der Zeile geloescht.

  H H loescht alle rechts vom Kursor stehenden Zeichen und geht bei I oder X in den Einfuegemodus ueber. Damit koennen vorteilhaft Anweisungen am Zeilenende ersetzt werden.

  K [i]K<ch> loescht alle Zeichen ab der aktuellen Kursorposition bis das i-te Vorkommen des Zeichens <ch> erreicht wird. Dieses Zeichen wird nicht mit geloescht. Die geloeschten Zeichen werden durch "\" begrenzt ausgegeben.

- Text suchen

  S Das Subkommando [i]S<ch> setzt den Kursor links vor das i-te Vorkommen des Zeichens <ch>. Wird das Zeichen <ch> nicht i-mal in der Zeile gefunden, bleibt der Kursor am Zeilenende stehen. Alle Zeichen, ueber die der Kursor bewegt wird, werden ausgegeben.

- Text ersetzen

  C Das Subkommando [i]C<text> ersetzt i Zeichen ab Kursorposition durch i Eingabezeichen (<text> muss genau die Laenge i haben!). Nach Eingabe der i Zeichen des neuen Textes wird ein neues EDIT-Subkommando erwartet.

- Neubeginn der Editierung

  L Das L-Subkommando rettet die bis dahin editierte Zeile durch Rueckspeichern der aufbereiteten Zeile in den Programmtext, gibt den rechts vom Kursor stehenden Teil aus und stellt den Kursor zurueck auf den Anfang der Zeile. L wird vornehmlich als erstes EDIT-Subkommando zum Ausgeben der aktuellen Zeile oder zur zwischenzeitlichen Orientierung nach laengeren Editierungssequenzen benutzt.

  A Das A-Subkommando rettet die editierte Zeile durch Rueckspeichern in den Programmtext und setzt den Kursor auf den Zeilenbeginn. Danach werden weitere EDIT-Subkommandos erwartet.

- Beenden der Editierung

  <CR> Bei Eingabe von <CR> wird der rechts vom Kursor stehende Teil der Zeile ausgegeben und die Zeile in das BASIC-Programm uebernommen. Der EDIT-Modus wird verlassen.

  E Wie <CR>, nur wird der rechts vom Kursor stehende Teil der Zeile nicht ausgegeben.

  Q Q bewirkt den Abbruch der Editierung. Alle eingegebenen Aenderungen der Zeile, die nicht vorher durch L oder A gerettet wurden, gehen verloren.

Fehler im EDIT-Modus

Auf unzulaessige Subkommandos, zu lange Zeilen usw. reagiert BASIC durch Ausgabe des Zeichens <CTRL/G>. Das betreffende Zeichen wird ignoriert.

EDIT-Modus nach Syntaxfehler

Der EDIT-Modus wird automatisch eingestellt, wenn eine Zeile abgearbeitet werden soll, die nicht den syntaktischen Konventionen von BASIC entspricht.

Beispiel

```
10  A(I(=J

RUN
---
?Syntax error in 10
10
```

Sofortiges Editieren

Stellt man bei der Eingabe einer Zeile einen Fehler fest und will die Zeile korrigieren, so kann der EDIT-Modus durch Eingabe von <CTRL/A> sofort betreten werden.
Anstelle einer Zeilennummer erscheint auf der neuen Zeile ein Ausrufungszeichen gefolgt von einem Leerzeichen. Der Kursor steht auf dem ersten Zeichen der Zeile. Es koennen EDIT-Kommandos gegeben werden.
Mit CTRL/U ist auch das Editieren von Eingabedaten moeglich.

Hinweis

Bei allen abgeschlossenen Veraenderungen am Programm geht die Moeglichkeit zur Programmfortsetzung mit CONT verloren. Variablen und Felder behalten gleichfalls ihre Werte nur, solange sich das Programm nicht aendert.
Sollen noch Werte von Variablen erhalten bleiben nachdem der EDIT-Modus (evtl. automatisch!) erreicht wurde, muss der EDIT-Modus mit Q verlassen werden, und zwar vor einem L- oder A-Subkommando.

5.2.4. LIST

Syntax

```
LIST[<zeilennummer>][-[<zeilennummer>]]
```

Semantik

Mit dem LIST-Kommando kann das gesamte aktuelle BASIC-Programm oder ein ausgewaehlter Programmbereich ueber das Bediengeraet ausgegeben werden.

<zeilennummer> ist eine ganze Zahl im Bereich von 0-65529.
Dabei gilt:

LIST <zeilennummer>	diese eine Zeile wird ausgegeben
LIST <zeilennummer>-	diese und alle folgenden (hoehernumerierten) Zeilen werden ausgegeben
LIST -<zeilennummer>	alle Programmzeilen vom Programmanfang bis einschliesslich der angegebenen Zeile werden ausgegeben
LIST <zeilennummer>-<zeilennummer>	alle Programmzeilen, die im angegebenen Zeilenbereich liegen, werden ausgegeben.

Fehlt die Angabe der Zeilennummer, so wird das gesamte BASIC-Programm ausgegeben.
Anstelle <Zeilennmmer> kann mit "." ein Bezug auf die aktuelle Zeile erfolgen. Das ist die Zeile, die zuletzt eingegeben oder editiert wurde bzw. die Zeile, bei der ein Programmlauf wegen eines Fehlers oder nach einer STOP-Anweisung unterbrochen wurde.
Nach LIST kehrt BASIC in das Kommandoniveau zurueck.

BEISPIEL

LIST 130-2000 alle Programmzeilen, deren Zeilennummer im angegebenen Zahlenbereich liegen, werden ueber das Terminal ausgegeben
LIST das aktuelle sich im Arbeitsspeicher befindende Programm wird ausgegeben

5.2.5. LLIST

Syntax

LLIST [<zeilennummer>][-[<zeilennummer>]]

Semantik

Das aktuelle BASIC-Programm bzw. die angegebenen Programmzeilen werden ueber den Drucker ausgegeben. Alle anderen Konventionen entsprechen denen des LIST-Kommandos. Nach LLIST kehrt BASIC in den Kommandomodus zurueck.

5.2.6. RENUM

Syntax

RENUM [<zeilennummer neu>][,[<zeilennummer alt>]
[,<schrittweite>]]

Semantik

Mit der RENUM-Anweisung kann das gesamte Programm oder nur ein Programmteil neu durchnumeriert werden.
<zeilennummer neu> ist die erste Zeilennummer, die vergeben werden soll. Fehlt diese Angabe, so wird standardmaessig 10 angenommen.
<zeilennummer alt> gibt die Zeile des Programmes an, bei welcher mit der Umnumerierung begonnen werden soll. Fehlt diese Angabe so wird bei der ersten Programmzeile mit der Umnumerierung angefangen.
<schrittweite> gibt den Abstand zwischen zwei aufeinanderfolgenden Zeilennummern der neuen Folge an. Fehlt diese Angabe, so wird standardmaessig 10 angenommen.

Bei der Neunumerierung werden alle Zeilennummernreferenzen des alten Programms (nach GOTO-, GOSUB-, THEN-, ON...GOTO-, ON...GOSUB- und ERL-Anweisungen) im neuen Programm wieder hergestellt. Treten dabei noch undefinierte Referenzen auf, so werden diese in der folgenden Form als Fehlermeldung ausgegeben:

"undefined line xxxxx in yyyyy"

xxxxx ist eine undefinierte Referenz des alten Programmes
yyyyy kann bereits eine neunumerierte Programmzeile sein

Die RENUM-Anweisung darf nicht die logische Folge der Programmzeilen aendern. Ebenso duerfen die neuen Zeilennummern nicht groesser als 65529 werden. Solche unzulaessigen RENUM-Anweisungen werden mit einer Fehlermeldung quittiert. Nach RENUM kehrt BASIC in den Kommandomodus zurueck.

Beispiel

RENUM Das gesamte Programm wird neu numeriert. Die erste Programmzeile erhaelt die Zeilennummer 10. Die Zeilennummern werden in Zehnerschritten vergeben.

5.2.7. NEW

Syntax

NEW

Semantik

Durch das NEW-Kommando wird der Arbeitsspeicher des BASIC-Systems, in dem sich das aktuelle BASIC-Programm und alle Variablen befinden, geloescht. Das NEW-Kommando ist zu geben, bevor ein neues BASIC-Programm eingegeben wird. Das BASIC-System kehrt in den Kommandomodus zurueck.

5.3. Programmausfuehrungskommandos

5.3.1. CLEAR

Syntax

CLEAR [,[<speicherende>][,<stack>]]

Semantik

CLEAR loescht alle numerischen Variablen auf den Wert Null, alle Stringvariablen auf die leere Zeichenkette und schliesst die offenen Dateien.
Fuer <speicherende> kann ein Ausdruck angegeben werden, der die hoechste von BASIC benutzbare Adresse definiert (siehe Schalter /M: beim Start des Interpreters).
Fuer <stack> kann ein Ausdruck angegeben werden, der die Groesse des von BASIC benutzten Stacks definiert.

5.3.2. CONT

Syntax

CONT

Semantik

Mit dem CONT-Kommando kann die Programmabarbeitung nach Programmunterbrechung durch <CTRL/C> oder STOP-Anweisung oder nach Programmende (END-Anweisung) fortgesetzt werden. Die Programmabarbeitung wird an der Unterbrechungsstelle fortgesetzt. Wird bei der Abarbeitung einer INPUT-Anweisung nach der Eingabeanforderung unterbrochen, wird die Eingabeanforderung bei Programmfortsetzung wiederholt.
Das CONT-Kommando wird zusammen mit entsprechenden STOP-Anweisungen zum Programmtest verwendet. Nachdem die Programmabarbeitung unterbrochen worden ist, koennen Zwischenergebnisse im Direktmodus abgefragt oder veraendert werden. Die Programmabarbeitung wird dann entweder mit CONT oder mit einer GOTO-Anweisung im Direktmodus fortgesetzt.
Programmfortsetzung ist auch nach Fehlern moeglich.
Die Programmfortsetzung mit CONT ist unzulaessig, wenn das Programm nach der Unterbrechung geaendert wurde.

Beispiel

(s. Abschn. 6.6.2., STOP-Anweisung)

5.3.3. RUN

Syntax

RUN [<zeilennummer>]

oder

RUN <dateispezifikation>[,R]

Semantik

Das RUN-Kommando veranlasst die Abarbeitung eines BASIC-1700-Programms. Es gibt zwei Formen:

- Durch ein RUN-Kommando der 1. Form wird ein BASIC-Programm, das sich bereits im Arbeitsspeicher befindet, gestartet. Wird nach dem Schluesselwort RUN eine Zeilennummer angegeben, so beginnt die Programmausfuerung auf dieser Zeile. Sonst beginnt die Programmabarbeitung auf der ersten Programmzeile.

- Durch ein RUN-Kommando der 2.Form wird ein BASIC-Programm, das unter der angegebenen Dateispezifikation vorliegt, in den Arbeitsspeicher geladen und anschliessend abgearbeitet. Vor dem

Laden des Programms werden alle offenen Dateien geschlossen und der Arbeitsspeicher geloescht. Wird jedoch die Option "R" angegeben, so bleiben alle Dateien geoeffnet. Enthaelt die Dateispezifikation keine explizite Geraeteangabe, so wird das augenblicklich aktuelle Geraet angenommen. Ist kein Dateityp spezifiziert, so wird .BAS eingesetzt.

BASIC kehrt nach Ausfuehrung des RUN-Kommandos zum Kommandoniveau zurueck.

5.3.4. TRON/TROFF

Syntax

```
TRON

TROFF
```

Semantik

Zur Unterstuetzung der Fehlersuche besitzt das BASIC-System eine einfache TRACE-Moeglichkeit. Das TRON (TRACE-ON)-Kommando setzt das TRACE-Flag. Bei gesetztem TRACE-Flag wird die Zeilennummer jeder abgearbeiteten Programmzeile auf dem Bedienterminal angezeigt. Die Zeilennummern werden in eckige Klammern eingeschlossen.
Durch das TROFF (TRACE-OFF)-Kommando oder ein NEW-Kommando wird das TRACE-Flag zurueckgesetzt. TRON und TROFF arbeiten im direkten oder indirekten Operationsmodus.

Beispiel

```
TRON
LIST
10 SUM%=0
20 FOR I%=1 TO 3
30 SUM%=SUM%+I%
40 PRINT I%;SUM%
50 NEXT
60 END

RUN
[10] [20] [30] [40] 1 1
[50] [20] [30] [40] 2 3
[50] [20] [30] [40] 3 6
[50] [60]
TROFF
```

5.3.5. System

Syntax

SYSTEM

Semantik

System schliesst alle Dateien und uebergibt die Steuerung dem Betriebssystem. Im Gegensatz wird mit CTRL/C nur das Kommandoniveau des BASIC-Systems erreicht.

Hinweis

SYSTEM kann auch als letzte Anweisung in einem BASIC-Programm stehen. Es ist dann sinnvoll, wenn nach Beendigung der Abarbeitung eines BASIC-Programms die Steuerung wieder dem Betriebssystem uebergeben werden soll.

5.4. Diskettenbezogene Kommandos

5.4.1. KILL

Syntax

KILL <dateispezifikation>

Semantik

Die KILL-Anweisung wird zum Loeschen aller Typen von Diskettendateien verwendet:
- Programmdateien
- Direktzugriffsdateien
- sequentielle Dateien

Die KILL-Anweisung darf keinesfalls auf geoeffnete Dateien angewendet werden, die noch gelesen oder beschrieben werden sollen.

Beispiel

```
100 KILL "EINDAT"
```

5.4.2. LOAD

Syntax

LOAD <dateispezifikation>[,R]

Semantik

Die Programmdatei mit der angegebenen Dateispezifikation wird in den Arbeitsspeicher eingelesen. Die Programmdatei ist zu einem frueheren Zeitpunkt mit einem SAVE-Kommando erstellt worden. Ist kein Geraet spezifiziert, so wird das momentan aktuelle Geraet angenommen. Fuer einen fehlenden Dateityp wird .BAS eingesetzt.
Durch ein LOAD-Kommando ohne R-Option werden vor dem Laden der angegebenen Programmdatei alle geoeffneten Dateien geschlossen und alle Variablen und Programmzeilen, die sich im Arbeitsspeicher befinden, geloescht.
Ist im LOAD-Kommando die R-Option gesetzt, so bleiben alle offenen Dateien geoeffnet. Das neue Programm wird in den Arbeitsspeicher geladen und anschliessend abgearbeitet. LOAD mit der R-Option kann zum Verketten mehrerer Programme bzw. Programmteile verwendet werden, die auf die gleichen Dateien zugreifen.

Hinweis

LOAD <Dateispezifikation>,R und
RUN <Dateispezifikation>,R haben die gleiche Wirkung.

Beispiel

```
LOAD "PROG2",R
```

5.4.3. MERGE

Syntax

MERGE <dateispezifikation>

Semantik

Die unter der Dateispezifikation vorliegende Programmdatei wird durch das MERGE-Kommando satzweise eingelesen und in das sich im Arbeitsspeicher befindende aktuelle Programm einsortiert.
Fuer das Einsortieren gilt:

- Treten Programmzeilen mit gleichen Zeilennummern auf, so werden die alten Programmzeilen durch die neuen Programmzeilen, die von der Diskette gelesen werden, ersetzt.
  Die einzugebende Programmdatei muss im KOI-7-Code vorliegen, sonst erfolgt eine Fehlermeldung (siehe SAVE-Kommando Option A).
- Enthaelt die Dateispezifikation keine Geraeteangabe,so wird das augenblicklich aktuelle Geraet genommen. Fehlt die Typ-Angabe, der Datei, so wird .BAS eingesetzt.

- Nach Ausfuehrung eines MERGE-Kommandos kehrt BASIC in den Kommandomodus zurueck.

5.4.4. NAME

Syntax

```
NAME <dateispezifikation alte Datei> AS <dateispezifikation
      neue Datei>
```

Semantik

Eine unter dem alten Dateinamen gespeicherte Datei erhaelt den neuen Dateinamen.
Durch NAME wird lediglich eine Namensaenderung im Dateiverzeichnis vorgenommen, die alte Datei wird nicht kopiert. Existiert unter dem alten Dateinamen keine Datei auf der Diskette oder ist der neue Dateiname bereits vergeben, so erfolgt eine Fehlermeldung. Fehlen in der Dateispezfikation der alten Datei Typ- und/oder Geraeteangabe, so werden fuer den Typ ".BAS" und fuer das Geraet das momentan aktuelle Diskettengeraet angenommen. Fehlen Typ- und/oder Geraeteangabe in der Dateispezifikation der neuen Datei, so werden diese aus der Dateispezifikation der alten Datei uebernommen.

Hinweis

Die NAME-Anweisung darf nicht auf geoeffnete Dateien angewendet werden.

Beispiel

```
NAME "A:FKT.BAS" AS "AUFG"
```

Die Datei FKT.BAS auf dem Diskettengeraet A: erhaelt den Namen AUFG.BAS.

5.4.5. SAVE

Syntax

```
SAVE <dateispezifikation>[,<option>]
```

Fuer Option kann A oder P stehen.

Semantik

Das aktuelle Programm wird unter der angegebenen Dateispezifikation auf die Diskette ausgegeben.
Existiert unter dem angegebenen Dateinamen bereits eine Datei, so wird diese ueberschrieben.
Ist kein Geraet spezifiziert, so wird das gerade aktuelle Diskettengeraet angenommen.

Fehlt der Dateiname oder ist dieser laenger als acht Zeichen, so erfolgt eine Fehlermeldung und das Kommando SAVE wird nicht ausgefuehrt.
Ist kein Dateityp angegeben, so wird .BAS angenommen. Wahlweise kann nach dem Dateinamen eine Option A oder P angegeben werden. Ist keine Option angegeben, so wird das Programm in einem verdichteten binaeren Format gespeichert.
Die Option A bewirkt, dass die Programmdatei als Folge von KOI-7-Code-Zeichen gespeichert wird. Dieses Textformat benoetigt zwar mehr Speicherplatz, jedoch fordert das MERGE-Kommando, dass die Eingabedatei im KOI-7-Code vorliegt.
Die Option P bewirkt, dass die Programmdatei in einem verschluesselten binaeren Kode gespeichert wird. Solche Programmdateien gelten als geschuetzt, da sie nicht mehr in die Originalform zurueckverwandelt werden koennen. Die Programme koennen nur mit LOAD und RUN geladen und abgearbeitet werden. Jeder Versuch solch ein Programm mit EDIT zu editieren oder mit LIST auszugeben, wird mit der Fehlermeldung "Illegal Function call" quittiert.

5.4.6. RESET

Syntax

RESET

Semantik

RESET ist nach dem Wechsel von Disketten anzuwenden, wenn anschliessend auf diese Daten ausgegeben werden sollen.
RESET ersetzt die Funktion von CTRL/C nach einem Diskettenwechsel unter dem Betriebssystem SCP, d.h. das Diskettensystem wird initialisiert.

Hinweis

Bevor ein Diskettenwechsel erfolgt muss gesichert sein, dass alle auf der Diskette gespeicherten Dateien ordentlich geschlossen sind. Zum Schliessen der Dateien ist das CLOSE-Kommando zu verwenden.

6. Anweisungen

6.1. Kommentaranweisung REM

Syntax

REM <text>

oder

' <text>

Semantik

Die Anweisung REM wird zur Kommentarisierung der BASIC-Programme verwendet.
Dazu gibt es drei Moeglichkeiten:

- Kommentaranweisung als letzte Anweisung auf Mehranweisungszeilen (:REM...).
- Einfuegen ganzer Kommentarzeilen an beliebigen Stellen des BASIC-Programms. Die Kommentarzeilen werden durch das Schluesselwort REM oder durch ein Apostroph eingeleitet.
- Anhaengen von Kommentaren am Ende von Anweisungszeilen. Dem Kommentartext ist in diesem Fall ein Apostroph voranzustellen.

Kommentare werden bei der Programmbearbeitung ignoriert, erscheinen jedoch bei der Ausgabe des Programmtextes. Sie koennen mit einer Ausnahme an jeder beliebigen Stelle des Programms eingefuegt werden. Unmittelbar nach einer DATA-Anweisung ist das Einfuegen von durch Apostroph gekennzeichnetem Kommentartext nicht moeglich, da dieser als Element der Datenliste gewertet wuerde.

Beispiel

```
100 'Beispiel
110 REM SUMMIERUNG
120 FOR J=1 TO N
130 SUMME = SUMME + VEKTOR(J)
140 NEXT J
150 DURCH = SUMME/N      'BERECHNETE DURCHSCHNITT
 .
 .
 .
```

6.2. Typdefinitionsanweisungen DEFINT, DEFSNG, DEFDBL, DEFSTR

Syntax

DEF<typ> <bereichsangabe> [,<bereichsangabe>...]

<typ>::= INT,SNG,DBL,STR

Semantik

Mit Typdefinitionsanweisungen koennen Variablen ueber ihre Anfangsbuchstaben bestimmte Typen zugeordnet werden.
Diese Anweisungen beziehen sich nur auf Variablennamen, deren Typ nicht schon durch ein angehaengtes Zeichen %, ¤, # festgelegt ist. In der Bereichsangabe (s. Beispiel) werden Buchstaben aufgelistet. Variablen, deren Namen mit einem der aufgelisteten Buchstaben beginnt, sind vom nach DEF notierten Typ.
Variablen, deren Typ weder durch eine Definitionsanweisung noch durch ein angehaengtes Typdeklarationszeichen festgelegt sind, sind vom Typ Gleitpunkt, einfache Genauigkeit.
Erfolgen fuer einen Anfangsbuchstaben nacheinander mehrere Typdefinitionen, so gilt die zuletzt abgearbeitete. Typdeklarationsanweisungen gehoeren an den Programmanfang.

Beispiel

Ein Programm enthalte die folgenden Typdefinitionsanweisungen:

```
10 DEFINT I-K,X
20 DEFDBL A
```

Fuer alle im Programm auftretenden Variablen, deren Namen nicht mit einem Typdeklarationszeichen enden, gilt:

- Variablen, deren Namen mit I, J, K oder X anfangen, sind vom Typ Integer,
- Variablen, deren Namen mit A anfangen, sind vom Typ numerisch, doppelte Genauigkeit und
- alle anderen Variablen sind vom Typ numerisch, einfache Genauigkeit.

6.3. Wertzuweisungen

6.3.1. LET

Syntax

[LET] <variable> = <ausdruck>

Semantik

Der Wert des Ausdruckes wird der Variablen als Wert zugewiesen.
Das Schluesselwort LET ist wahlweise anzugeben.

Beispiele

```
100 LET A=2 : LET B=3
110 LET C=A^2 + B^2
  .
  .
  .
```

entspricht der Anweisungsfolge

```
100 A=2 : B=3
110 C=A^2 + B^2
```

6.3.2. SWAP

Syntax

SWAP <variable>, <variable>

Semantik

Die Werte der angegebenen Variablen werden getauscht. Dabei muessen die Variablen vom gleichen Typ sein, sonst erfolgt die Fehlermeldung "Type mismatch". Alle vier Variablentypen sind moeglich.

Beispiel

```
10 A¤= "HENRY " : B¤= "HAS " : C¤= "MONEY"
20 PRINT A¤+B¤+C¤+" !"
30 SWAP A¤,B¤
40 PRINT A¤;B¤ C¤ " ?"

RUN
---
HENRY HAS MONEY !
HAS HENRY MONEY ?
```

6.3.3. MID¤

Syntax

MID¤ (<zeichenkettenausdruck 1>,<n>[,<m>]) =
<zeichenkettenausdruck 2>

<n> und <m> sind Integerausdruecke

Semantik

Eine Teilzeichenkette wird durch eine andere ersetzt. Die Zeichen des Zeichenkettenausdruckes 1 werden ab der Zeichenposition n durch m Zeichen des Zeichenkettenausdruckes 2 ueberschrieben. Mit m kann angegeben werden, wie viele Zeichen des Zeichenkettenaus-

druckes 2 zum Ueberschreiben verwendet werden sollen. Fehlt m, so werden alle Zeichen von Zeichenkettenausdruck 2 uebernommen. Die aktuelle Laenge des Zeichenkettenausdruckes 1 wird dabei nicht veraendert.

Beispiel

```
10 A¤= "TAG: MO"
20 MID¤ (A¤,6,2)= "DIENSTAG"
30 PRINT A¤

RUN
TAG: DI
```

Hinweis

MID¤ gibt es auch als Standardfunktion (s. Abschn. 7.6.).

6.3.4. RANDOMIZE

Syntax

```
RANDOMIZE [<numerischer ausdruck>]
```

Semantik

Die RANDOMIZE-Anweisung bewirkt eine erneute Initialisierung des Zufallszahlengenerators mit dem Wert des angegebenen numerischen Ausdruckes. Ist kein numerischer Ausdruck angegeben, so wird bei Abarbeitung der RANDOMIZE-Anweisung der Bediener durch die Ausschrift

```
RANDOM NUMBER SEED (-32768 to 32767)?
```

aufgefordert, einen Initialisierungswert einzugeben. Enthaelt ein Programm keine RANDOMIZE-Anweisung, so wird bei jedem Programmlauf die gleiche Folge von Zufallszahlen erzeugt. Steht am Anfang des Programms eine RANDOMIZE-Anweisung, so werden in jedem Programmlauf durch unterschiedliche Initialisierung des Zufallszahlengenerators verschiedene Zahlenfolgen erzeugt.

Beispiel

```
10 WEITER¤= "J"
20 WHILE WEITER¤= "J"
30 RANDOMIZE
40 FOR I=1 TO 5
50 PRINT RND;
60 NEXT
65 PRINT
70 INPUT "WEITER J/N";WEITER¤
80 WEND
```

```
RUN
---
RANDOM NUMBER SEED (-32768 TO 32767)? 300
                                      ---
.9196374 .5186567 .421009 .8032628 .8260937

WEITER J/N? J
            -
RANDOM NUMBER SEED (-32768 TO 32767)? 5
                                     -
.537536E-02 .9370679 .8688921 .3502141 .5133648

.
.
.
```

6.4. Dialog- Ein-/Ausgabe

6.4.1. INPUT

Syntax

```
INPUT [;][<"ausschrift">;]<variablenliste>
```

Semantik

Durch eine INPUT-Anweisung koennen waehrend der Programmabarbeitung ueber das Bediengeraet Daten eingegeben werden. Die Programmausfuehrung wird unterbrochen, und auf dem Bediengeraet wird durch Ausgabe eines Fragezeichens angezeigt, dass eine Eingabe erwartet wird.
Enthaelt die INPUT-Anweisung eine Ausschrift, so erscheint diese vor dem Fragezeichen auf dem Terminal. Die Ausgabe einer Ausschrift ermoeglicht es dem Programmierer, die einzugebenden Daten naeher zu beschreiben. Nach der Ausschrift kann anstatt des Semikolons auch ein Komma stehen. Es erscheint dann kein Fragezeichen nach der Ausschrift.
Folgt dem Schluesselwort INPUT ein Semikolon, so wird ein durch den Nutzer eingegebenes "carriage return" nicht auf dem Terminal ausgegeben.
Die eingegebenen Daten werden den Variablen, so wie sie in der Variablenliste aufgefuehrt sind, der Reihe nach zugewiesen. Die Anzahl der eingegebenen Daten muss mit der Anzahl der spezifizierten Variablen in der Variablenliste uebereinstimmen. Die einzelnen Dateneinheiten werden durch Kommas getrennt.
In der Variablenliste koennen numerische Variablen und Zeichenkettenvariablen (einschliesslich indizierte Variablen) stehen. Der Typ der eingegebenen Dateneinheit muss dem Typ der spezifizierten Variablen entsprechen. Einzugebende Zeichenketten brauchen nicht in Anfuehrungsstriche eingeschlossen werden.

Hinweis

Erfolgen Eingaben mit zu viel oder zu wenig Dateneinheiten, oder die Datentypen stimmen nicht ueberein, erscheint die Fehlermel-

dung "?REDO FROM START". Die Eingabe muss vollstaendig wiederholt werden. Bei fehlerhaften Eingaben werden keine Wertzuweisungen vorgenommen.

Beispiele

```
100 PI=3.14
200 INPUT "EINGABE DES RADIUS R = ",R
250 IF R=0 THEN END
300 A=PI*R^2
400 PRINT "KREISFLAECHE=";A
500 PRINT
600 GOTO 200

RUN
EINGABE DES RADIUS R = 5.0
KREISFLAECHE= 78.5

EINGABE DES RADIUS R =
.
.
.

10 INPUT "EINGABE Y,Z";Y,Z
20 X=Y/Z
30 PRINT "X = ";X
40 END

RUN
EINGABE Y,Z? 30,-15
X = -2
```

6.4.2. LINE INPUT

Syntax

```
LINE INPUT [;][<"ausschrift">];<zeichenkettenvariable>
```

Semantik

Die LINE-INPUT-Anweisung wird zur Eingabe einer ganzen Zeile ueber das Bediengeraet verwendet. Sie wird als Zeichenkettenvariable interpretiert und der spezifizierten Zeichenkettenvariablen als Wert zugeordnet. Jede ueber das Bediengeraet einzugebende Zeile wird angefordert (Kursorstand).
Ist in der LINE-INPUT-Anweisung eine Ausschrift angegeben, so erscheint diese vor der Eingabeanforderung auf dem Bediengeraet. Es wird im Gegensatz zur INPUT-Anweisung kein gesondertes Fragezeichen ausgeschrieben, wenn es nicht Bestandteil der Ausschrift ist. Ein einzugebender Datensatz wird durch das Zeichen "carriage return" beendet. Ein Datensatz kann maximal 254 Zeichen umfassen. Das Zeichen "line feed" (Fortsetzung der Eingabezeile) wird in

die Zeichenkette aufgenommen.
Folgt direkt nach dem Schluesselwort LINE INPUT ein Semikolon, so wird bei Eingabe eines "carriage return" durch den Bediener der Uebergang zu einer neuen Eingabezeile auf dem Bediengeraet unterdrueckt.
Eine fehlerhafte Eingabezeile kann durch Eingabe von <CTRL/C> annulliert werden. Nach Eingabe des Kommandos CONT kann die Eingabe wiederholt werden.

Hinweis

Fuehrende Leerzeichen, Begrenzungszeichen usw. sind Bestandteil der Zeichenkette.

Beispiel

Unterschiede zwischen INPUT und LINE INPUT

```
10 INPUT A¤                    10 LINE INPUT A¤
20 PRINT A¤;"*"                20 PRINT A¤;"*"
30 PRINT "**********"          30 PRINT "**********"

RUN                            RUN
---                            ---
?     ABC                          ABC
  ---------                    ---------
ABC*                               ABC  *
**********                     **********
```

Die fuehrenden Leerzeichen gehoeren mit zur Zeichenkette. Die Eingabeanforderung ohne Fragezeichen wird nur durch Kursorstand angezeigt.

6.4.3. PRINT

Syntax

```
PRINT [<liste von ausdruecken>]    oder
  ?   [<liste von ausdruecken>]
```

Semantik

Die PRINT-Anweisung wird zur Ausgabe von Daten ueber das Bediengeraet verwendet. Folgt nach PRINT bzw. ? kein Ausdruck, so wird eine Leerzeile ausgegeben. Anderenfalls werden die Werte der aufgefuehrten Ausdruecke ueber das Bediengeraet ausgegeben.
Die in der Liste enthaltenen Ausdruecke koennen numerische und/oder Zeichenkettenausdruecke sein. Zeichenketten muessen in Anfuehrungsstriche (") eingeschlossen sein.
Die Druckpositionen der einzelnen Datenelemente werden durch die angegebenen Trenner zwischen den Ausdruecken in der Ausgabeliste bestimmt. Jede Ausgabezeile ist in Druckzonen zu je 14 Zeichen unterteilt. Als Trenner sind die Zeichen Komma, Semikolon und Leerzeichen zugelassen.
Ein Komma bewirkt den Uebergang zum Anfang der naechsten Druck-

zone. Ein Semikolon bewirkt, dass das folgende Datenelement unmittelbar hinter dem vorhergehenden steht. Anstelle eines Semikolons koennen auch ein oder mehrere Leerzeichen zwischen den Ausdruecken stehen. Die Anordnung der auszugebenden Datenelemente kann auch mit Hilfe der TAB-Funktion gesteuert werden (s. Abschn. 7.8.). Wird eine Ausdrucksliste durch ein Komma, Semikolon oder Leerzeichen abgeschlossen, so wird die naechste PRINT-Ausgabe auf gleicher Zeile fortgesetzt. Sonst erfolgt der Uebergang zu einer neuen Ausgabezeile.
Ueberschreitet eine Ausgabezeile die Bildschirmbreite, so wird die Ausgabe der Datenelemente auf der folgenden Zeile fortgesetzt.
Die Ausgabe von numerischen Werten erfolgt in der beschriebenen Form:

- Das Vorzeichen wird unmittelbar vor der Zahl ausgegeben.
- Anstelle eines positiven Vorzeichens erscheint ein Leerzeichen.
- Auf jede auszugebende Zahl folgt ein Leerzeichen.

Numerische Werte einfacher Genauigkeit werden, solange sie mit 7 oder weniger Ziffern ohne Genauigkeitsverlust dargestellt werden koennen, in einfacher Form, sonst im E-Format ausgegeben.

Beispiel

```
1E-7          ->   .0000001
1E-8          ->   1E-08
```

Numerische Werte doppelter Genauigkeit werden solange sie mit 16 oder weniger Ziffern ohne Genauigkeitsverlust gegenueber der Exponentendarstellung dargestellt werden koennen, in einfacher Form, ansonsten im E-Format ausgegeben.

Beispiele

```
1D-16         ->   .0000000000000001
1D-17         ->   1D-17
```

```
10 PRINT "TABELLE 5: ZWEIERPOTENZEN"
20 PRINT "--------------------------"
30 PRINT
40 PRINT "N","2^N","2^-N"
50 FOR N%=0 TO 10
60 PRINT N%, 2^N%, 2^-N%
70 NEXT
```

```
RUN
---
TABELLE 5: ZWEIERPOTENZEN
-------------------------

N               2^N             2^-N
 0               1               1
 1               2               .5
 2               4               .25
 3               8               .125
 4               16              .0625
 5               32              .03125
 6               64              .015625
 7               128             .0078125
 8               256             3.90625E-03
 9               512             1.953125E-03
 10              1024            9.765625E-04
```

In diesem Beispiel wird gezeigt, wie Kommas zweckmaessig eingesetzt werden, um eine Reihe von Werten spaltengerecht in einem Druckbild anzuordnen. Durch die PRINT-Anweisung in Zeile 30 wird eine Leerzeile ausgegeben.

```
10 FOR I=1 TO 3
20 PRINT I,
30 NEXT
40 PRINT
50 FOR I=1 TO 3
60 PRINT I;
70 NEXT I
80 PRINT

RUN
---
 1               2               3

 1  2  3
```

Hier wird die Wirkung von Kommas und Semikolon gezeigt, wenn diese die Ausdrucksliste abschliessen. Die beiden PRINT-Anweisungen in den Zeilen 40 und 80 dienen dem Abschluss der in der vorangehenden Laufanweisung aufgebauten Datensaetze. Durch sie werden demzufolge keine Leerzeilen ausgegeben.

6.4.4. PRINT USING

Syntax

```
PRINT USING <formatbeschreibung>;<liste von ausdruecken>
```

Semantik

Die PRINT USING-Anweisung dient der formatierten Ausgabe von Daten ueber das Bediengeraet.
Die Liste von Ausdruecken enthaelt die durch Semikolon getrennten Formatbeschreibungen und numerischen Ausdruecke, deren Werte ausgegeben werden sollen.

Die Formatbeschreibung ist eine Zeichenkettenkonstante oder eine Zeichenkettenvariable, bestehend aus speziellen Formatzeichen. Solche Folgen von Formatzeichen stellen Aufbereitungsvorschriften fuer Zeichenketten oder numerische Werte dar.

Aufbereitungsvorschriften fuer Zeichenketten

Eines von den drei folgenden Formatzeichen kann zur Beschreibung des Ausgabefeldes benutzt werden:

"!"	Nur das erste Zeichen der Zeichenkette wird gedruckt.
"\nLeerzeichen\"	2+n Zeichen der Zeichenkette sollen gedruckt werden. Die restlichen Zeichen der Zeichenkette werden ignoriert. "\\" zwei Zeichen werden gedruckt "\ \" drei Zeichen werden gedruckt Ist das spezifizierte Ausgabefeld laenger als die angegebene Zeichenkette, so wird die Zeichenkette linksbuendig ins Ausgabefeld eingetragen und die restlichen Zeichenpositionen werden mit Leerzeichen aufgefuellt.
"&"	Die Laenge des Ausgabefeldes entspricht der Laenge der auszugebebden Zeichenkette, d.h. die Laenge der Zeichenkette ist variabel.

Beispiel

```
10 A¤="GUTEN" : B¤="TAG" : FORMAT¤="\ \"
20 PRINT USING "!";"HALLO","PAUL"
25 PRINT USING FORMAT¤;A¤
30 PRINT USING "\   \";A¤;B¤
40 PRINT USING "\    \";A¤;B¤;"PAUL"
50 PRINT USING "&";A¤;" ";B¤;" !"

RUN
---
HP
GUT
GUTENTAG
GUTEN TAG   PAUL
GUTEN TAG !
```

Aufbreitungsvorschriften fuer numerische Werte

Die folgenden Formatzeichen koennen zur Formatierung des Ausgabefeldes benutzt werden. Jedes Zeichen der Aufbereitungsvorschrift reserviert Platz fuer ein Zeichen im Druckbild.

Das Zeichen "#" repraesentiert eine Ziffernstelle. Wird eine Ziffernstelle wertmaessig nicht belegt, so erscheint im Druckbild entweder ein Leerzeichen oder eine Null als Ersatzzeichen.
In die Folge der Ziffernzeichen "#" kann ein Dezimalpunkt eingefuegt werden. Fehlt der Dezimalpunkt, so wird dieser

hinter der letzten Ziffernstelle angenommen (ganze Zahl).
Werden durch "#" markierte Ziffernstellen links vom Dezimalpunkt wertmaessig nicht belegt, so werden diese im Druckbild durch Leerzeichen ersetzt. Nur die unmittelbar links vor dem Komma stehende Ziffernstelle wird gegebenenfalls durch eine Null ersetzt.
Alle durch "#" markierten Ziffernstellen, die rechts vom Dezimalpunkt stehen, werden durch Null ersetzt, wenn sie wertmaessig nicht belegt werden koennen. Sind rechts vom Dezimalpunkt zu wenig Ziffernstellen vorgesehen, so wird der auszugebende Wert gerundet.

Beispiel

```
PRINT USING "##.### ";.91;3.121;66.5552;5.4
 0.910  3.121 66.555  5.400
```

Im Beispiel wurde am Ende der Aufbereitungsvorschrift ein Leerzeichen eingefuegt, um die Ausgabewerte voneinander zu trennen.

\+ Das Pluszeichen kann am Anfang oder Ende der Aufbereitungsvorschrift stehen. Es gibt an, ob das Vorzeichen (+/-) des auszugebenden Wertes vor oder nach der Zahl stehen soll.

Beispiel

```
PRINT USING "+##.##    ";-90.42;3.4;-.6;56.6
-90.42     +3.40     -0.60    +56.6
```

\- Das Minuszeichen kann am Ende der Aufbereitungsvorschrift stehen. Es bewirkt, dass negative Werte mit einem nachstehenden Minus ausgegeben werden. An positive Zahlen wird ein Leerzeichen angehaengt.

Beispiel

```
PRINT USING "##.##-    ";-90.42;3.4;-.6;56.6
90.42-     3.40      0.60-    56.6
```

** Zwei aufeinanderfolgende Stern-Zeichen koennen am Anfang einer Formatierungsvorschrift stehen. Sie spezifizieren zwei Ziffernstellen und dienen als Fuellzeichen, wenn Ziffernstellen links vom Dezimalpunkt durch den auszugebenden Wert nicht belegt werden koennen.

Beispiel

```
PRINT USING "**##.#    ";12.3;-0.9;3120.;510.5;1.1"
**12.3    **-0.9    3120.0    *510.5    ***1.1
```

¤¤ Zwei aufeinanderfolgende Dollarzeichen koennen am Anfang der Formatierungsvorschrift stehen. Sie bewirken, dass fuehrende Nullen durch Leerzeichen ersetzt werden und vor der ersten signifikanten Ziffer ein Dollarzeichen ausgegeben wird. Durch ¤¤ werden zwei Ziffernpositionen spezifiziert, wovon

eine durch das Dollarzeichen selbst belegt wird.
Die Angabe von ¤¤ ist nicht beim Exponentialformat moeglich.
Negative Zahlen koennen nur ausgegeben werden, wenn das Minuszeichem an das Ende der Zeichenreihe geschoben wird.

Beispiel

```
PRINT USING "¤¤###.##    ";456.78;51.22;3333.44
 ¤456.78      ¤51.22    ¤3333.44
```

¤ Durch Angabe von "¤" am Anfang der Formatisierungsvorschrift wird die kombinierte Wirkung von "**" und "¤¤" erreicht.
Fuehrende Nullen erscheinen als "*" und links vor dem aufbereiteten Wert steht "¤". Durch "**¤" werden drei Ziffernpositionen markiert, eine von ihnen wird zum Dollarzeichen.

Beispiel

```
PRINT USING "**¤##.##  ";2.56;7000.99;40099;55555.55
***¤2.56  ¤7000.99  *¤400,99  %¤55555.55
```

Der letzte Wert in diesem Beispiel kann von der Maske nicht aufgenommen werden und ist dashalb durch ein % gekennzeichnet (siehe Hinweis).

, Ein Komma, das links vom Dezimalpunkt in der Formatierungsvorschrift steht, bewirkt, dass links vom Dezimalpunkt nach jeweils drei Ziffern ein Komma eingeschoben wird. Jedes eingeschobene Komma belegt eine Ziffernstelle und ist in der Formatierungsvorschrift vorzusehen.
Ein Komma am Ende der Formatierungsvorschrift wird als Bestandteil des aufbereiteten Ausgabefeldes mitgedruckt. Das Komma bringt im Exponentialformat keinen Effekt.

Beispiele

```
PRINT USING "#######,.##    ";701290.30;9000000.70
701,290.30   %9,000,000.70
```

Der letzte Wert kann mit der Maske nicht aufbereitet werden (Kennzeichnung durch %). Es fehlen fuer die einzuschiebenden Kommas zwei Ziffernpositionen.

```
PRINT USING "####.###,";4012.300;52.1
4012.300,  52.100,
```

^^^^ Vier aufeinanderfolgende Potenzierungszeichen, die hinter den markierten Ziffernstellen angegeben werden koennen, bewirken die Aufbereitung des numerischen Wertes im Exponentialformat. Anstelle von "^^^^" in der Formatisierungsvorschrift erscheint im Druckbild die Angabe von "E+XX" bzw. "E-XX". Die signifikanten Ziffernstellen des aufzubereitenden Wertes werden linksbuendig den markierten Ziffernstellen zugeordnet, und der Exponent wird dementsprechend bereitgestellt. Falls kein vor- oder nachgestelltes Vorzeichen spezifiziert ist, wird immer eine Ziffernposition links vom

Dezimalpunkt zur Ausgabe des Vorzeichens verwendet.

Beispiele

```
PRINT USING "##.##^^^^  ";3478.92;-8691
 3.48E+03  -8.69E+03

PRINT USING ".####^^^^- ";777777;-3333
 .7778E+06 -.3333E+04

PRINT USING "+##.##^^^^";-666.66
-66.67E+01
```

Ein Unterstreichungsstrich in der Formatierungsvorschrift bewirkt, dass das darauffolgende Zeichen im Druckbild als "Literalzeichen" erscheint.

Beispiel

```
PRINT USING " ?##.## ?";33.44
?33.44?
```

Das "Literalzeichen" kann selbst ein Unterstreichungsstrich sein.

```
PRINT USING "  ##.##  ";33.44
 33.44
```

Hinweis

Kann der auszugebende Wert nicht durch die spezifizierten Ziffernpositionen in der Formatierungsvorschrift aufgenommen werden, so erscheint im Druckbild vor der auszugebenden Zahl ein Prozentzeichen "%". Das geschieht auch, wenn ein numerischer Wert durch Rundung zu gross wird.

Beispiel

```
PRINT USING "##.##    ";221.11;99.999
%221.11    %100.00
```

Werden mehr als 24 Ziffernstellen spezifiziert, so erfolgt die Fehlermeldung "illegal function call".

6.4.5. LPRINT und LPRINT USING

Syntax

```
LPRINT [<liste von ausdruecken>]

LPRINT USING <formatbeschreibung>;<liste von ausdruecken>
```

Semantik

Die LPRINT- und LPRINT-USING-Anweisungen arbeiten wie die PRINT- bzw. PRINT-USING-Anweisungen, nur dass alle Ausgaben ueber den Drucker erfolgen. Es wird ein Drucker mit einer Druckbreite von 132 Zeichen vorausgesetzt.

6.4.6. WRITE

Syntax

WRITE [<liste von ausdruecken>]

Semantik

Die Werte der in der Liste aufgefuehrten Ausdruecke werden ueber das Terminal ausgegeben. In der Liste koennen Zeichenkettenausdruecke und numerische Ausdruecke enthalten sein, die durch Kommas getrennt sind. Bei der Ausgabe werden die einzelnen Datenelemente durch Kommas voneinander getrennt. An das letzte auszugebende Datenelement wird automatisch ein "carriage return""line feed" angehangen.
Zeichenketten werden in Anfuehrungsstriche eingeschlossen. Die Ausgabeformate fuer numerische Werte entsprechen denen der PRINT-Anweisung (s. Abschn. 2.4.3.).
Ist nach dem Schluesselwort WRITE kein Ausdruck angegeben, so wird eine Leerzeile ausgedruckt.

Beispiel

```
10 A=60:B=-70:C¤="zeichenkette":D#=12D-9
15 E!=12E-9
20 WRITE A,B,C¤,D#,E!

RUN
---
60,-70,"zeichenkette",.000000012,1.2E-08
```

6.4.7. WIDTH

Syntax

WIDTH <numerischer ausdruck>

Semantik

Der numerische Ausdruck legt die Ausgabebreite des Bildschirms fest. Als Ausgabebreite sind nur 36,40 oder 80 Zeichen zugelassen. Beim Wechsel der Ausgabebreite durch WITDH wird der Bildschirm geloescht. Wird die durch WIDTH definierte Ausgabebreite ueberschritten, so wird automatisch ein "carriage return""line feed" an die Zeile angefuegt.

6.5. Arbeit mit programminternen Daten

6.5.1. DATA

Syntax

DATA <liste von konstanten>

Semantik

Die in einem Programm enthaltenen DATA-Anweisungen formieren eine programminterne Datenliste. Die in dieser Datenliste gespeicherten Daten koennen in der Reihenfolge, wie die DATA-Anweisungen im Programm auftreten, durch READ-Anweisungen (s. Abschn. 6.5.2.) gelesen werden.
Die Zahl der Konstanten in einer DATA-Anweisung ist nur durch die maximale Zeilenlaenge begrenzt. Die Konstantenliste kann numerische Konstanten (Festpunktzahlen, Gleitpunktzahlen, Integerzahlen) oder Zeichenkettenkonstanten enthalten. Zeichenkettenkonstanten koennen in der DATA-Anweisung ohne begrenzende Anfuehrungsstriche stehen, wenn sie keine Kommas, Doppelpunkte oder signifikante fuehrende oder nachfolgende Leerzeichen enthalten. Sind diese Zeichen enthalten, muss die Zeichenkettenkonstante in Anfuehrungsstriche eingeschlossen werden.
Der Variablentyp in einer READ-Anweisung muss dem Typ der zu lesenden Konstante entsprechen. Ein wiederholtes Lesen der Datenliste kann nach Ausfuehrung einer RESTORE-Anweisung (s. Abschn. 6.5.3.) erfolgen. DATA-Anweisungen sind nicht ausfuehrbar und werden bei der Programmabarbeitung ohne Wirkung uebergangen.

Beispiel

(s. Abschn. 6.5.2. READ-Anweisung)

6.5.2. READ

Syntax

READ <liste von variablen>

Semantik

Durch READ-Anweisungen werden die mit einer DATA-Anweisung gespeicherten Konstanten der Reihe nach gelesen und den spezfizierten Variablen zugewiesen.
Dabei muss der Typ der in der READ-Anweisung aufgefuehrten Variablen dem Typ der zu lesenden Konstanten entsprechen. Sonst erfolgt eine Fehlermeldung.
Die Variablenliste kann Zeichenkettenvariablen oder numerische Variablen enthalten.
Mit einer READ-Anweisung koennen mehrere Datenlisten gelesen werden. Ebenso koennen die Daten einer Datenliste durch mehrere

READ-Anweisungen gelesen werden. Die Daten werden in der Reihenfolge, wie die DATA-Anweisungen im Programm auftreten, durch READ-Anweisungen gelesen.
Falls die Anzahl der in der Variabenliste enthaltenen Variablen groesser ist als die Anzahl der durch die DATA-Anweisung(en) vereinbarten Datenelemente, so erfolgt die Fehlermeldung "OUT OF DATA".
Sollen die Datenelemente erneut von Anfang an gelesen werden, so ist dies durch Einfuegen einer RESTORE-Anweisung (s. Abschn. 6.5.3.) moeglich.

Beispiel

```
10 FOR I=0 TO 4
20 READ A(I),B(I)
30 PRINT A(I),B(I)
40 NEXT I
50 DATA 1,2,3,4,5
60 DATA 6,7,8,9,10
70 END
```

RUN

```
 1             2
 3             4
 5             6
 7             8
 9             10
```

6.5.3. RESTORE

Syntax

RESTORE [<zeilennummer>]

Semantik

Eine RESTORE-Anweisung ohne Zeilennummer bewirkt, dass bei der naechsten READ-Anweisung auf das erste Datenelement, das in der ersten DATA-Anweisung des Programms vereinbart wurde, zugegriffen wird.
Ist in einer RESTORE-Anweisung eine Zeilennummer angegeben, so greift die naechste READ-Anweisung auf das erste Datenelement, das in der durch die Zeilennummer spezifizierten DATA-Anweisung vereinbart wurde, zu.

Beispiel

```
10 READ A,B,C: PRINT A,B,C
20 RESTORE
30 READ D,E,F: PRINT D,E,F
40 DATA 10,20,30
```

```
RUN
---
10          20          30
10          20          30
```

6.6. Steueranweisungen

6.6.1. END

Syntax

```
END
```

Semantik

Die END-Anweisung bewirkt das Beenden des laufenden BASIC-Programms, das Schliessen aller noch geoeffneten Dateien und die Rueckkehr zum BASIC-Kommandoniveau.
END-Anweisungen koennen an beliebiger Stelle im Programm auftreten. Am physischen Programmende kann die END-Anweisung wahlweise angegeben werden.
Soll ein Programm nur kurzzeitig unterbrochen werden, um Werte von Variablen abzufragen, bzw. zu modifizieren, so ist die STOP-Anweisung zu benutzen (s. Abschn. 6.6.2.).

Beispiel

```
90 IF VAR<10E-6 THEN END ELSE GOTO 320
```

6.6.2. STOP

Syntax

```
STOP
```

Semantik

Durch die STOP-Anweisung wird die Programmabarbeitung unterbrochen. STOP-Anweisungen koennen an beliebiger Stelle im Programm stehen. Wird eine STOP-Anweisung erreicht, so erscheint auf dem Teminal die Ausschrift:

```
BREAK IN LINE nnnnn
```

Im Unterschied zur END-Anweisung werden durch die STOP-Anweisung die Dateien nicht geschlossen. Nach Abarbeitung einer STOP-Anweisung kehrt das BASIC-System in den Kommandostatus zurueck. Die Programmabarbeitung kann durch ein CONT-Kommando an der Unterbrechungsstelle fortgesetzt werden (s. Abschn.5.3.2.).

6.6.3. GOTO

Syntax

```
GOTO <zeilennummer>
```

Semantik

Die Programmabarbeitung wird bei der ersten ausfuehrbaren Anweisung, die auf der spezifizierten Programmzeile bzw. auf deren Folgezeilen steht, fortgesetzt.

Beispiel

```
10 REM ABBRUCH DES ZYKLUS DURCH DATENENDE
20 READ X
30 PRINT "X =";X,
40 Q=X^2
50 PRINT "QUADRAT =";Q
60 GOTO 10
70 DATA 2,3,10

RUN
 X = 2          QUADRAT = 4
 X = 3          QUADRAT = 9
 X = 10         QUADRAT = 100

OUT OF DATA IN 20
```

Hinweis

Auf Zeile 10 steht keine ausfuehrbare Anweisung, die Programmabarbeitung wird auf Zeile 20 mit der READ-Anweisung fortgesetzt.

6.6.4. GOSUB...RETURN

Syntax

```
GOSUB <zeilennummer>

RETURN
```

Semantik

Durch die GOSUB-Anweisung wird die Verzweigung zu einem BASIC-Unterprogramm realisiert. Nach GOSUB wird das Verzweigungsziel, d.h. die erste Zeilennummer des Unterprogramms angegeben.
Vor der Verzweigung wird die Adresse der Anweisung, welche auf die GOSUB-Anweisung folgt, als Rueckkehradresse festgelegt. Die Rueckkehr aus einem Unterprogramm erfolgt durch die RETURN-Anweisung. Eine RETURN-Anweisung innerhalb eines Unterprogramms bewirkt den Ruecksprung zu der Anweisung, die der zuletzt abgearbeiteten GOSUB-Anweisung folgt. Eine RETURN-Anweisung darf nur

abgearbeitet werden, wenn zuvor eine GOSUB- bzw. ON GOSUB-Anweisung abgearbeitet wurde, sonst erfolgt eine Fehlermeldung. Ein Unterprogramm kann jedoch mehrere RETURN-Anweisungen enthalten. Es ist moeglich, Unterprogramme ineinander zu schachteln.

Beispiel

```
10 REM ---HAUPTPROGRAMM---
20 PRINT "HAUPTPROGRAMM RUFT UNTERPROGRAMM 1"
30 GOSUB 100
40 PRINT "AUS UNTERPROGRAMM 1 ZURUECKGEKEHRT"
50 PRINT "ENDE"
60 END
100 REM ---UNTERPROGRAMM 1---
110 PRINT TAB(5)"UNTERPROGRAMM 1 RUFT UNTERPROGRAMM 2"
120 GOSUB 200
130 PRINT TAB(9)"AUS UNTERPROGRAMM 2 ZURUECKGEKEHRT"
140 RETURN
200 REM ---UNTERPROGRAMM 2---
210 PRINT TAB(5)"UNTERPROGRAMM 2 ABGEARBEITET"
220 RETURN

RUN
---
HAUPTPROGRAMM RUFT UNTERPROGRAMM 1
    UNTERPROGRAMM 1 RUFT UNTERPROGRAMM 2
        UNTERPROGRAMM 2 ABGEARBEITET
    AUS UNTERPROGRAMM 2 ZURUECKGEKEHRT
AUS UNTERPROGRAMM 1 ZURUECKGEKEHRT
ENDE
```

6.6.5. ON...GOSUB und ON...GOTO

Syntax

```
ON <ausdruck> GOTO <liste von zeilennummern>

ON <ausdruck> GOSUB <liste von zeilennummern>
```

Semantik

Die ON...GOTO- und ON...GOSUB-Anweisungen ermoeglichen das Verzweigen zu verschiedenen Programmzeilen eines BASIC-Programmes. Der nach ON angegebene Ausdruck liefert als Wert einen Verzweigungsindex I, gemaess dem die I-te Zeilennummer aus der Liste der Zeilennummern ausgewaehlt wird. Anschliessend wird eine GOTO- bzw. GOSUB-Anweisung zu der so ermittelten Zeilennummer durchgefuehrt (s. Abschn. 6.6.3. und 6.6.4.).
Fuer den ermittelten Verzweigungsindex I gilt:
- Ergab der Ausdruck einen REAL-Wert, so wird I in einen INTEGER-Wert konvertiert.
- Ist I<0 oder I>255, so erfolgt eine Fehlermeldung.
- Ist I=0 oder I ist groesser als die Anzahl der in der Liste aufgefuehrten Zeilennummern, so wird die Programmabarbeitung bei der naechsten ausfuehrbaren BASIC-Anweisung fortgesetzt.

Die Zeilennummernliste darf maximal 255 durch Komma getrennte

Zeilennummern enthalten.

Hinweis

Die nach GOSUB aufgefuehrten Zeilennummern muessen jeweils erste Zeilennummern von Unterprogrammen sein.

Beispiel

```
100  ON X GOTO 50,20,200
105
 .
 .
200
```

Abarbeitung der Programmzeile 100:
wenn X=1, Sprung nach Zeile 50 (1.Zeilennummer in der Liste)
wenn X=2, Sprung nach Zeile 20 (2.Zeilennummer in der Liste)
wenn X=3, Sprung nach Zeile 200 (3.Zeilennummer in der Liste)
wenn X<0, Fehlermeldung
wenn X>255, Fehlermeldung
wenn X=0, Abarbeitung auf Zeile 105 wird fortgesetzt
wenn X>3, Abarbeitung auf Zeile 105 wird fortgesetzt

6.6.6. FOR...NEXT

Syntax

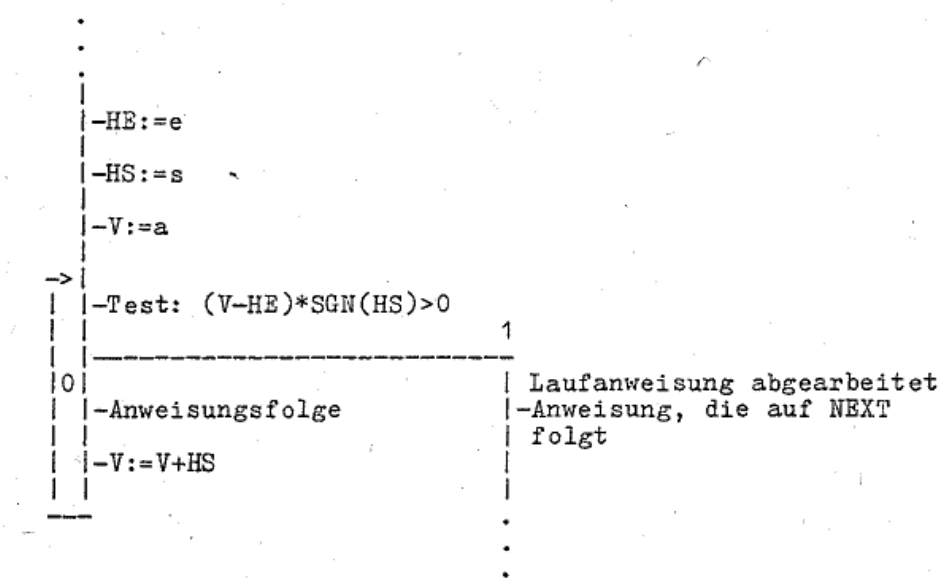
```
FOR <variable>= <a> TO <e> [STEP <s>]
 .
 .
 .
[<anweisungsfolge>]
 .
 .
 .
NEXT [<variable>]
```

<a>,<e>,<s> sind numerische Ausdruecke
<variable> einfach genaue Variable oder Intergervariable

Semantik

FOR-Anweisung, Anweisungsfolge und NEXT-Anweisung sind als Einheit zu betrachten und werden als Laufanweisung bezeichnet. Die nach FOR angegebene Variable wird als Laufvariable (oder Zaehler) verwendet. In der FOR-Anweisung werden fuer die Laufvariable Anfangswert (a), Schrittweite (s) und Endwert (e) festgelegt.
Die Schrittweite ist der Wert, um welchen die Laufvariable nach jedem Durchlauf der Anweisungsfolge zu veraendern ist. Die Schrittweite kann einen beliebigen positiven oder negativen Wert annehmen. Sie sollte nie den Wert Null annehmen,da sonst eine endlose Schleife entsteht.
Fehlt die Angabe der Schrittweite, so wird standardmaessig der Wert 1 angenommen.

Fuer eine Laufanweisung gilt folgende Abarbeitungsfolge:



HE und HS sind Hilfsvariablen, V ist Laufvariable.
Der Endwert der Laufvariable und die Schrittweite werden vor dem Anfangswert berechnet.
Endwert und Schrittweite werden vor Eintritt in den Laufbereich der Laufanweisung berechnet und koennen nicht durch die Anweisungsfolge veraendert werden.

Hinweis

Wird die Schrittweite durch eine REAL-Zahl dargestellt, so sind Rundungsfehler, die den Abbruchtest beeintraechtigen koennen, zu beachten.

Beispiel

```
10 FOR I=1 TO 2.0 STEP 0.1
20 PRINT I;
30 NEXT

RUN
---
 1  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9
```

Da sich der Wert 0.1 nicht genau darstellen laesst, wird die Laufanweisung infolge Rundungsfehler nur 10 mal statt 11 mal abgearbeitet.

Schachtelung von Laufanweisungen

FOR...NEXT-Anweisungen koennen ineinander verschachtelt werden. Jede Laufanweisung muss eine eigene Laufvariable besitzen, d.h. die Namen der Laufvariablen muessen sich voneinander unterscheiden. Fehlerhafte Programme mit verschachtelten Laufanweisungen,

in denen eine Laufvariable mehrmals auftritt, laufen endlos. Die innere Laufanweisung muss jeweils vor der aeusseren Laufanweisung abgeschlossen werden. Haben geschachtelte Laufanweisungen den gleichen Endpunkt, so koennen diese durch eine kompakte NEXT-Anweisung folgender Form abgeschlossen werden:

NEXT <variable i>,<variable i-1>...<variable 1>

Dies ist gleichbedeutend mit der Anweisungsfolge:

NEXT <variable i>:NEXT <variable i-1>...:NEXT <variable 1>

wobei <variable i> der Zaehler der innersten Laufanweisung sein muss. Auf jede FOR-Anweisung muss eine NEXT-Anweisung folgen. Die Variable nach NEXT kann fehlen. Die NEXT-Anweisung bezieht sich dann auf die letzte FOR-Anweisung.

Beispiele

```
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
```

```
RUN
---
 1  2  3  4  5  6  7  8  9  10
```

Der Endwert der Laufanweisung wird berechnet, bevor der Anfangswert der Laufvariablen zugewiesen wird.

```
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
```

Die Laufanweisung wird nicht abgearbeitet, da der Anfangswert der Laufvariable bereits groesser ist als der Endwert.

```
10 J=0
20 FOR I=1 TO J STEP -1
30 PRINT I;
40 NEXT
```

```
RUN
---
 1  0
```

In diesem Fall wird die PRINT-Anweisung zweimal durchlaufen.

Beispiel fuer eine geschachtelte Laufanweisung

```
10 DIM FELD(1,2)
20 FOR I=0 TO 1
30 FOR J=0 TO 2
40 PRINT "FELD(";I;",";J;")=";
50 INPUT FELD(I,J)
60 NEXT J,I
```

```
RUN
---
FELD(0,0)=1
          -
FELD(0,1)=2
          -
FELD(0,2)=3
          -
FELD(1,0)=4
          -
FELD(1,1)=5
          -
FELD(1,2)=6
          -
```

In zwei ineinander geschachtelten Laufanweisungen werden den Elementen des zweidimensionalen Feldes Werte zugewiesen. Die beiden Laufanweisungen haben einen gemeinsamen Endpunkt und koennen deshalb durch eine kompakte NEXT-Anweisung abgeschlossen werden.

6.6.7. WHILE...WEND

Syntax

```
WHILE <ausdruck>
  .
  .
[<anweisungen>]
  .
  .
WEND
```

Semantik

Fuer die WHILE...WEND-Schleifenanweisung gilt folgende Abarbeitungsfolge:

Schritt 1: Liefert <ausdruck> den Wert Null (falsch), dann weiter bei Schritt 3.
Liefert <ausdruck> einen Wert ungleich Null (wahr), dann weiter bei Schritt 2.

Schritt 2: Ausfuehrung der Anweisungsfolge, die zwischen WHILE und WEND steht.
Beim Erkennen des Schluesselworts WEND, Uebergang zu Schritt 1.

Schritt 3: Schleifenanweisung ist abgeschlossen. Die Programmausfuehrung wird bei der naechsten Anweisung, die dem Schluesselwort WEND folgt, fortgesetzt.

WHILE...WEND-Schleifen koennen beliebig ineinander geschachtelt werden. Das Schluesselwort WEND wird jeweils dem letzten (innersten) WHILE zugeordnet.
Ein WHILE ohne dazugehoeriges WEND liefert den Fehler "WHILE without WEND". Ein WEND ohne vorangegangenes WHILE liefert den Fehler "WEND without WHILE".

Beispiel

```
100  REM Bubble-Sort-Zyklus fuer Feld A mit J% Elementen
110  WEITER%=1: N%=J%
120  WHILE WEITER%
130        WEITER%=0: N%=N%-1
140        FOR I%=1 TO N%
150            IF A(I%)>A(I%+1)
               THEN SWAP A(I%),A(I%+1): WEITER%=1
160        NEXT I%
170  WEND
```

6.6.8. IF...THEN[...ELSE] und IF..GOTO[...ELSE]

Syntax

```
IF <ausdruck> THEN <anweisung(en)>/<zeilennummer>
   [ELSE<anweisun(en)>/<zeilennummer>]

IF <ausdruck> GOTO <zeilennummer>
   [ELSE<anweisung(en)>/<zeilennummer>
```

Semantik

Liefert der <ausdruck> den logischen Wert "wahr" bzw. einen Wert ungleich Null, so wird der THEN- oder GOTO-Zweig abgearbeitet.
Liefert <ausdruck> den logischen Wert "falsch" bzw. den Wert Null, so wird der THEN- bzw. GOTO-Zweig ignoriert und, falls vorhanden, der ELSE-Zweig abgearbeitet. Die Programmabarbeitung wird anschliessend bei der naechsten Anweisung auf der folgenden Programmzeile fortgesetzt. Dem Schluesselwort THEN koennen ein oder mehrere Anweisungen folgen, die auszufuehren sind oder eine Zeilennummer, zu der verzweigt werden soll.
Nach dem Schluesselwort GOTO muss die Zeilennumer angegeben werden, bei welcher die Abarbeitung fortzufuehren ist. Vor dem Schluesselwort THEN kann ein Komma stehen.
IF-Anweisungen koennen ineinander verschachtelt werden. Dabei wird die Verschachtelungstiefe nur durch die zulaessige Zeilenlaenge begrenzt. Zum Beispiel ist

```
IF I>J THEN PRINT "GROESSER" ELSE (line feed)
   IF I<J THEN PRINT "KLEINER" ELSE PRINT "GLEICH"
```

eine zulaessige Anweisung.

Werden unvollstaendige IF-Anweisungen verschachtelt, d.h. die Anzahl der THEN- und ELSE-Zweige ist unterschiedlich, so ist folgendes zu beachten.
Die Zuordnung der ELSE-Zweige zu den entsprechenden THEN-Zweigen erfolgt von innen nach aussen.
In der folgenden Anweisung stehen THEN und zugehoeriges ELSE untereinander.

```
10 IF EPS < E-3 THEN IF ITER > 100 THEN 250 (line feed)
                                   ELSE 260 (line feed)
                ELSE N = N+1: GOTO 180
```

Hinweis

Wird die IF..THEN..ELSE Anweisung im Direktmodus verwendet, so darf sie keine Zeilennummer enthalten.
Werden in Verzweigsbedingungen arithmetische Werte, die aus einer Gleitkommaberechnung hervorgegangen sind getestet, so sind Ungenauigkeiten, welche sich aus der internen Datendarstellung ergeben koennen, zu beachten.

Beispiele

```
100 IF A<0 THEN PRINT "A NEGATIV":GOTO 300
200 A=A-1
300 ...


100 IF A>=0 THEN A=A-1 ELSE PRINT "A NEGATIV"
200 ...
```

Diese beiden IF-Anweisungen realsieren den gleichen Sachverhalt.

```
240 IF I% THEN 640
```
Falls I% ungleich Null ist, wird zur Zeile 640 verzweigt.

6.7. Dimensionen von Feldern

6.7.1. DIM

Syntax

```
DIM <liste indizierter variablen>
```

Semantik

Durch die Abarbeitung einer DIM-Anweisung koennen die Anzahl und der Umfang der Dimensionen von Feldern festgelegt werden. Durch die DIM-Anweisung wird festgelegt, wieviel Speicherplatz die einzelnen Felder benoetigen. Felder koennen eine oder mehrere Dimensionen besitzen. Der Umfang einer Dimension wird durch eine untere und eine obere Grenze beschrieben. Die untere Indexgrenze ist standardmaessig 0. Sie kann mit der OPTION-BASE-Anweisung fuer alle Felder auf 1 gesetzt werden (siehe 6.7.2.). Ein Feld, dessen Indexgrenzen nicht mit DIM festgelegt wurden, hat in allen Dimensionen die obere Indexgrenze 10. Erfolgt spaeter ein Feldzugriff mit einem Index, der ausserhalb der definierten Indexgrenzen fuer dieses Feld liegt, so wird der Fehler "Subscipt out of range" gemeldet. Die DIM-Anweisung loescht alle Feldelemente auf Null (bzw. leere Zeichenkette).

Beispiel

```
10  DIM DIAG(3,3,3)
20  FOR I=0 TO 3
30  FOR J=0 TO 3
40  FOR K=0 TO 3
50  PRINT "DIAG(";I;",";J;",";K;")=";
60  INPUT DIAG(I,J,K)
70  NEXT K,J,I
```

Das Feld DIAG ist 3-dimensional und hat pro Dimension 4 Elemente.

6.7.2. OPTION BASE

Syntax

```
OPTION BASE 0

OPTION BASE 1
```

Semantik

Mit Hilfe von OPTION BASE kann die untere Indexgrenze fuer die Dimensionierung von Feldern festgelegt werden. Standardmaessig wird als untere Grenze der Feldindizes der Wert 0 angenommen. Durch die Anweisung

```
OPTION BASE 1
```

wird als untere Grenze der Wert 1 eingestellt. Die OPTION-BASE-Anweisung muss vor der ersten DIM-Anweisung bzw. vor der ersten Anwendung einer indizierten Variablen im Programm abgearbeitet werden. Jedes Programm darf nur eine OPTION-BASE-Anweisung enthalten. Wird dies nicht beachtet, erfolgt eine Fehlermeldung.

Beispiel

```
10 OPTION BASE 1
20 DIM FELDA(10), FELDB(2,3)
```

FELDA hat 10 Elemente, FELDB hat 6 Elemente.
Wird die Zeile 10 aus dem Programm gestrichen, so gilt:
FELDA hat 11 Elemente, FELDB hat 12 Elemente.

6.7.3. ERASE

Syntax

```
ERASE <feldvariablenliste>
```

Semantik

Die in der Feldvariablenliste aufgefuehrten Felder werden geloescht. Ihr Speicherplatz wird frei und kann fuer andere Zwecke genutzt werden.
Unter gleichem Namen kann z.B. dann ein Feld mit anderen Grenzen durch Abarbeitung einer DIM-Anweisung neu errichtet werden. Versucht man die Neudimensionierung ohne zwischenzeitliches ERASE, wird der Fehler "Redimensioned array" gemeldet.

Beispiel

```
5 OPTION BASE 1
10 INPUT ("FELDUMFANG=");ANZ%
20 DIM FELD (ANZ%)
 .
 .
 .
200 ERASE FELD
210 GOTO 10
```

6.8. Anwendereigene Funktionsdefinition DEF FN

Syntax

DEF FN<funktionsname>[(<parameterliste>)]=<funktionsdefinition>

Semantik

Mit der DEF-FN-Anweisung kann der Benutzer eigene Funktionen definieren. Als <funktionsname> muss ein Variablenname angegeben werden. Zusammen mit den vorangestellten Zeichen FN bildet der angegebene Variablenname den Namen der Funktion. Die Wertart der Funktion wird durch den Namen der Funktion festgelegt.
In der Parameterliste werden die Namen der formalen Parameter der Funktion angegeben. Die hier angegebenen Parameter sind nur innerhalb der Funktionsdefinition gueltig und haben keine Beziehung zu gleichnamigen Variablen im umgebenden Programm.
Als <funktionsdefinition> ist ein Ausdruck anzugeben (numerisch oder string), der den Funktionswert liefert.
Die ganze Funktionsdefinition muss auf einer Zeile notiert werden. Rekursive oder indirekt rekursive Definition der Funktion sind nicht zulaessig und fuehren zum Fehler "out of memory". In der Funktionsdefinition vorkommende Variablennamen, die nicht in der Parameterliste vorkommen, beziehen sich auf Variable im umgebenden Programm.
Bevor eine Funktion erstmalig aufgerufen werden kann, muss ihre Definition abgearbeitet worden sein. Es ist sinnvoll, Funktionsdefinitionen am Programmbeginn zu notieren. Bei der Abarbeitung der Funktion werden den formalen Variablen die Werte der angegebenen aktuellen Parameter zugewiesen. Erforderlichenfalls erfolgen Typanpassungen. Dann wird der die Funktion definierende Ausdruck berechnet. Das Ergebnis wird, wenn noetig, dem im Funktionsnamen enthaltenen Typ angepasst.

Hinweis

Funktionsdefinitionen duerfen nicht im Direktmodus erfolgen.

Beispiel

```
110  DEF FNROSE(X1,X2)=(X1^2+X2)^2
 .
 .
1330  T=FNROSE(XF,XG)
```

6.9. Programmueberlagerung

6.9.1. CHAIN

Syntax

```
CHAIN [MERGE]<dateiname>[,[<zeile>][,ALL][,DELETE<bereich>]]
```

Semantik

CHAIN wird zum Starten eines BASIC-Programms benutzt oder zum Ueberlagern von Programmteilen. Der Dateiname bezeichnet das zu startende Programm bzw. den zu ladenden Programmteil.
Das mit dem Dateinamen spezifizierte Programm muss im externen Textformat vorliegen (s. SAVE).
Fuer <zeile> kann ein Ausdruck angegeben werden, der die Nummer der Zeile liefert, bei der das gerufene Programm gestartet wird. Fehlt der Ausdruck, wird bei der ersten Zeile gestartet.
Bei der Anweisung

```
CHAIN "prog1",1000
```

ist also 1000 eine numerische Konstante und wird nicht durch ein RENUM-Kommando beeinflusst.
Mit der Angabe ALL werden alle Variablen des aktuellen Programms dem zu startenden Programm uebergeben. Fehlt ALL, muessen alle zu uebergebenden Variablen in COMMON-Anweisungen aufgefuehrt werden.
Mit MERGE kann man Programmteile (z.B. ein Unterprogramm) in das aktuelle Programm einfuegen. Es wird wie beim MERGE-Kommando mit dem aktuellen Programm gemischt. Die einzusortierenden Programmteile muessen im KIO-7-Code vorliegen (siehe SAVE-Kommando Option A).

Liest man nacheinander verschiedene Programmteile auf diese Weise ein, dann empfiehlt es sich, den vorhandenen Programmteil vor Einlesen des naechsten Programmteils zu loeschen. Dazu gibt man DELETE <bereich> an. Die in <bereich> angegebenen Zeilennummern werden durch RENUM beeinflusst.

Beispiel

```
CHAIN "DONAR"                          startet Programm DONAR
CHAIN "CRTF",1020                      Start auf Zeile 1020
CHAIN "CRTF",1020,ALL                  zusaetzlich Uebergabe
                                       aller Variablen
CHAIN MERGE "OVERLAY",1000             Mischen des aktuellen Pro-
                                       gramms mit dem Programm
                                       "OVERLAY", Start auf Zei-
                                       le 1000
CHAIN MERGE "EVAL",DELETE 1000-3000    Mischen des aktuellen Pro-
                                       gramms mit dem Programm
                                       "EVAL". Vorher werden die
                                       Zeilen 1000-3000 im aktu-
                                       ellen Programm geloescht.
                                       Der Start erfolgt bei der
                                       1. Zeile des gemischten
                                       Programms.
```

Hinweis

Bei CHAIN MERGE bleiben eroeffnete Dateien offen, die aktuelle OPTION BASE wird nicht veraendert. Fehlt MERGE, so werden Typfestlegungen fuer Variable sowie nutzerdefinierte Funktionen nicht ins naechste Programm uebernommen. Alle DEFINT-, DEFSNG-, DEFDBL-, DEFSTR- und DEF-FN-Anweisungen, die sich auf von beiden Programmen benutzte Variablen beziehen, muessen im zu startenden Programm erneut angegeben werden.

6.9.2. COMMON

Syntax

```
COMMON <variablenliste>
```

Semantik

Die COMMON-Anweisung kennzeichnet Variablen, die an ein mit CHAIN zu startendes Programm uebergeben werden sollen. Variablen und Felder, die an ein mit CHAIN zu startendes Programm uebergeben werden sollen, werden mit Komma getrennt in der <variablenliste> aufgefuehrt. Felder werden durch "()" gekennzeichnet. Eine Variable bzw. ein Feld darf nur einmal als COMMON deklariert werden. Sollen alle Variablen und Felder an das zu startende Programm uebergeben werden, kann dafuer auch die ALL-Angabe in der CHAIN-Anweisung benutzt werden.
Obwohl COMMON-Anweisungen ueberall im Programm auftreten duerfen, ist es zweckmaessig, sie am Anfang des Programms nach den DIM-Anweisungen zu notieren.

Beispiel

```
100  COMMON A,B,C,D(),G%
110  CHAIN "PROG3",10
```

6.10. Fehlerbehandlung

6.10.1. ERROR

Syntax

ERROR <integerausdruck>

Semantik

Der Wert des Integer-Ausdrucks wird als Fehlercode gewertet. Er muss zwischen 0 und 255 liegen. Handelt es sich um einen Fehlercode, der von BASIC benutzt wird (siehe Anlage 1), so wird der entsprechende Fehler simuliert. Ist eine Fehlerroutine angemeldet, so wird zu dieser verzweigt, sonst wird die zugehoerige Fehlermeldung ausgegeben. Ist der Fehlercode nicht von BASIC belegt, so wird zu einer Fehlerroutine verzweigt, falls eine solche angemeldet ist (ON-ERROR-GOTO-Anweisung). Der Nutzer hat damit die Moeglichkeit eigene Fehlercodes einzufuehren und auszuwerten. Ist bei einem nicht von BASIC benutzten Fehlercode keine Fehlerroutine angemeldet, dann reagiert BASIC mit der "UNPRINTABLE ERROR" Meldung.

6.10.2. Die Variablen ERR und ERL

Die Variablen ERR und ERL werden in Fehlerbehandlungsroutinen verwendet. Wird eine mit der ON-ERROR-GOTO-Anweisung angemeldete Fehlerbehandlungsroutine erreicht, so enthaelt ERR eine Fehlernummer und ERL die Nummer der den Fehler verursachende Zeile. Mit den Werten dieser Variablen kann dann eine detaillierte Fehlermeldung erfolgen.

Hinweis

Fuehrt eine Direktmodus-Anweisung zum Eintritt in eine angemeldete Fehlerbehandlungsroutine des aktuellen Programms, so enthaelt ERL den Wert 65535.
Die Variablen ERL und ERR sind reservierte Variablen und duerfen nicht auf der linken Seite einer Ergibtanweisung stehen. In Vergleichen, in denen die Variable ERL mit einer Zeilennummer verglichen wird, muss die Zeilennummer rechts stehen. Links stehende Zeilennummern werden bei RENUM nicht neu numeriert.
Die Zuordnung von Fehlernummern zu Fehlerausschriften kann der Anlage 1 entnommen werden.
Die Anwendung der ERROR-Anweisung ermoeglicht bei der Programmtestung eine schnelle Zuordnung von Fehlernummern zur Fehlerausschrift.

6.10.3. ON ERROR GOTO

Syntax

ON ERROR GOTO <zeilennummer>

Semantik

Alle auftretenden Fehler ausser Zahlenueberlauf, Division durch Null und Zahlenunterlauf (s. Abschn. 4.2.) fuehren zur Verzweigung zu der durch <zeilennummer> angegebenen Behandlungsroutine. Existiert die angegebene Zeilennummer nicht, so erfolgt eine ON-ERROR-GOTO-0-Anweisung. Eine Fehlermeldung wird ausgegeben und die Abarbeitung unterbrochen. Es wird empfohlen, dass alle Fehlerbehandlungsroutinen eine ON-ERROR-GOTO-0-Anweisung ausfuehren, falls fuer einen entschluesselten Fehlercode keine vernuenftige Behandlung erfolgen kann.

Hinweis

Tritt innerhalb einer Fehlerbehandlungsroutine erneut ein Fehler auf, so wird die Abarbeitung mit einer Fehlermeldung beendet. Innerhalb einer Fehlerbehandlungsroutine darf kein Error-Trap erfolgen.

6.10.4. RESUME

Syntax

RESUME

RESUME 0

RESUME NEXT

RESUME <zeilennummer>

Semantik

Mit der RESUME-Anweisung kann die Programmausfuehrung nach der Abarbeitung einer Fehlerbehandlungsroutine fortgesetzt werden. Fuer die RESUME-Anweisung existieren vier Formate. Durch ein jedes Format wird bestimmt, wo die Programmabarbeitung fortzusetzen ist.

RESUME / RESUME 0	Die Programmabarbeitung wird bei der Anweisung, die den Fehler verursachte fortgesetzt.
RESUME NEXT	Die Programmabarbeitung wird bei der Anweisung, die unmittelbar auf die den Fehler verursachende Anweisung folgt, fortgesetzt.
RESUME <zeilennummer>	Die Programmabarbeitung wird auf der angegebenen Programmzeile fortgesetzt.

Eine RESUME-Anweisung darf nur innerhalb einer Fehlerbehandlungsroutine auftreten, sonst erfolgt eine Fehlermeldung.

Beispiel

```
10  ON ERROR GOTO 2000
20  FOR I=1 TO 256
30  A¤=A¤+"I"
40  NEXT I
50  PRINT LEN(A¤)
60  END
2000  PRINT ERL,ERR
2010  IF ERR=15 THEN RESUME 50

RUN
---
 30            15
 255
ok
NEW
---
ERROR 15                          (Im Direktmodus erfolgt Ausgabe
--------                          der entsprechenden Fehlerausschrift)
string too long
```

7. Standardfunktionen

7.1. Allgemeines

In diesem Kapitel werden mit wenigen Ausnahmen alle in BASIC vorhandenen Standardfunktionen vorgestellt. Standardfunktionen, die die Datei- und Unterprogrammarbeit unterstuetzen (EOF, LOF, LOC, INP, USR, PEEK und VARPTR) sind im Abschnitt ueber die Dateiarbeit beschrieben.

Syntax

```
funktionsname(argument[,argument])
```

Semantik

Standardfunktionen koennen von einem BASIC-Programm gerufen werden, ohne dass sie vorher definiert werden muessen.
Die Argumente von Funktionen sind stets in Klammern eingeschlossen.
In den folgenden Abschnitten werden die Argumente von Funktionen in folgender Weise dargestellt:

X und Y	repraesentieren beliebige numerische Ausdruecke
I und J	repraesentieren Integer-Ausdruecke
X¤ und Y¤	repraesentieren String-Ausdruecke

Wenn ein Gleitpunktwert an einer Stelle auftritt, an der ein Integer-Wert gefordert ist, wird der gebrochene Anteil gerundet und der resultierende Integer-Wert verwendet.

Tabelle 1: Standardfunktionen

Funktion		Parametertypen			Bedeutung
Name	Typ	P1	P2	P3	
SQR	real	real			Quadratwurzel
EXP	real	real			Exponentialfunktion
LOG	real	real			Logarithmus zur Basis E
INT	int	real			groesster Int-Wert des Argumentes
ABS	real	real			Absolutbetrag
FIX	int	real			ganzzahliger Teil des Argumentes
SGN	int	real			Signum
SIN	real	real			Sinus
COS	real	real			Cosinus
TAN	real	real			Tangens
ATN	real	real			Arcustangens

Tabelle 1: (Fortsetzung)

Funktion		Parametertypen			Bedeutung
Name	Typ	P1	P2	P3	
					Konvertierungen
ASC	int	string			Zeichen --> Int-Wert (KOI-7-Code)
VAL	real	string			Zeichenkette --> Real-Wert
CINT	int	real			Real-Wert --> Int-Wert
CDBL	real	real			Real-Wert --> doppelt genauer Wert
CSNG	real	real			Real-Wert --> einfach genauer Wert
OCT¤	string	real 1			Int-Wert dezimal --> Zeichenkette Oktalwert
HEX¤	string	real 1			Int-Wert --> Zeichenkette Hexadezimal-Wert
STR¤	string	real			Real-Wert --> Zeichenkette
CHR¤	string	int			Int-Wert --> Zeichen
					Zeichenkettenfunktionen
LEN	int	string			Laenge einer Zeichenkette
LEFT¤	string	string	int		linke Teilkette
RIGHT¤	string	string	int		rechte Teilkette
MID¤	string	string	int	int	mittlere Teilkette
INSTR	int	int	string	string	suchen Teilzeichenkette
SPACE¤	string	int			erzeugt Zeichenkette aus Leerzeichen
STRING¤	string	int	int/ string		erzeugt Zeichenkette lt. Parameter
INPUT¤	string	real 1	real 1		lesen Zeichenkette vom Terminal
INKEY¤	string				lesen Zeichenkette vom vom Terminal (ohne Echo)
RND	real	real			Zufallszahl
FRE	int	string real 2			Groesse des freien Speichers in Byte
TAB		int			Positionseinstellung im Ausgabedatensatz
SPC		int			Bereitstellung von Leerzeichen im Ausgabedatensatz
LPOS	int	int 2			aktuelle Position innerhalb Ausgabepuffer fuer Drucker
POS	int	int 2			aktuelle Kursorposition

1 wird vor Funktionsausfuehrung auf Int-Wert gerundet
2 Scheinargument

7.2. Algebraische Standardfunktionen

SQR(X)

Die Funktion liefert die quadratische Wurzel von X. X muss groesser oder gleich Null sein.

Beispiel

```
10  FOR X=10 TO 25 STEP 5
20  PRINT X,SQR(X)
30  NEXT

RUN
 10           3,162278
 15           3,872984
 20           4,472136
 25           5
```

EXP(X)

Die Funktion liefert den Wert e hoch X. Es muss X<=88.02969 gelten. Tritt ein Ueberlauf auf, so wird der Fehler "Overflow" gemeldet, die groesste darstellbare Zahl mit dem entsprechenden Vorzeichen als Resultat angenommen und die Programmabarbeitung fortgesetzt.

Beispiel

```
10  X=5 :Y=1
20  PRINT EXP(X-Y)

RUN
 54.59815
```

LOG(X)

Die Funktion liefert den natuerlichen Logarithmus von X. X muss groesser als Null sein.

Beispiel

```
PRINT LOG(71/9)
 2.065455
```

INT(X)

Die Funktion liefert den groessten Integer-Wert, der kleiner oder gleich X ist (siehe die Funktionen FIX und CINT, die ebenfalls Integer-Werte liefern).

Beispiel

```
PRINT INT(100.88);INT(-44.111)
 100 -45
```

ABS(X)

Die Funktion liefert den Absolutbetrag des Ausdruckes X.

Beispiel

```
PRINT ABS(-9*3)
 27
```

FIX(X)

Die Funktion liefert den ganzzahligen Teil von X. FIX(X) ist aequivalent zu SGN(X)*INT(ABS(X)). Der wichtigste Unterschied zwischen FIX und INT besteht darin, dass FIX fuer negative X nicht die naechstkleinere Zahl liefert.

Beispiel

```
PRINT FIX(21.81);FIX(-21.81)
 21 -21
```

SGN(X)

Die Funktion liefert den Wert
1 fuer X > 0,
0 fuer X = 0 und
-1 fuer X < 0.

Beispiel

```
10  ON SGN(X)+2 GOSUB 50,150,200
```

Wenn X negativ ist, wird zur Zeile 50,
bei X = 0 zur Zeile 150 und
fuer X > 0 zur Zeile 200 verzweigt.

7.3. Trigonometrische Standardfunktionen

SIN(X)

Die Funktion liefert den Sinus von X mit einfacher Genauigkeit. Der Winkel X ist im Bogenmass anzugeben.

Beispiel

```
PRINT SIN(2.3)
 .7457054
```

COS(X)

Die Funktion liefert den Cosinus von X mit einfacher Genauigkeit. Der Winkel X ist im Bogenmass anzugeben.

Beispiel

```
10 X=2.3
20 PRINT SIN(X)/COS(X)

RUN
 -1.119214
```

TAN(X)

Die Funktion liefert den Tangens von X mit einfacher Genauigkeit. Der Winkel X ist im Bogenmass anzugeben. Tritt ein Ueberlauf auf, so wird ein Fehler gemeldet, die groesste darstellbare Zahl mit dem entsprechenden Vorzeichen als Resultat angenommen und die Programmabarbeitung fortgesetzt.

Beispiel

```
PRINT TAN(2.3)
-1.119214
```

ATN(X)

Die Funktion liefert den Arcustangens von X im Bogenmass. Das Resultat liegt im Bereich von -pi/2 bis pi/2. Die Berechnung von ATN wird in einfacher Genauigkeit ausgefuehrt.

Beispiel

```
10  INPUT X
20  PRINT ATN(X)

RUN
? 3
 1.249046
```

7.4. Konvertierungsfunktionen, die einen numerischen Wert liefern

ASC(X¤)

Die Funktion liefert den KOI-7-Code des ersten Zeichens der Zeichenkette X¤ als numerischen Wert. Wenn X¤ leer ist, wird der Fehler "Illegal function call" gemeldet (siehe auch die CHR¤-Funktion zur Konvertierung eines KOI-7-Codes in ein Zeichen).

Beispiel

```
10  X¤="TAG"
20  PRINT ASC(X¤)

RUN
---
 84
```

VAL(X¤)

Die Funktion konvertiert die in der Zeichenkette X¤ dargestellte numerische Konstante in einen numerischen Wert. Die VAL-Funktion uebergeht fuehrende Leerzeichen, Tabulatoren und "line feed" in der Argumentzeichenkette (siehe auch die STR¤-Funktion).

Beispiel

```
10  INPUT X¤
20  PRINT VAL(X¤)

RUN
---
? "   -7.89"
-7.89
```

CINT(X)

Die Funktion konvertiert X in einen Integer-Wert durch Runden des gebrochenen Anteils. Wenn X nicht im Bereich von -32768 bis 32767 liegt, wird der Fehler "Overflow" gemeldet.

Beispiel

```
PRINT CINT(81.55)
 82
```

CDBL(X)

Die Funktion konvertiert X in einen doppelt genauen Wert.

Beispiel

```
PRINT CDBL(23.2513)
 23.25129890441895
```

CSNG(X)

Die Funktion konvertiert X in einen einfach genauen Wert (siehe die Funktionen CINT und CDBL zur Konvertierung von Werten in Integer bzw. doppelt genaue Werte).

Beispiel

```
PRINT CSNG(82.7812539#)
 82.78126
```

7.5. Konvertierungsfunktionen, die einen Zeichenkettenwert liefern

OCT¤(X)

Die Funktion liefert eine Zeichenkette, die den oktalen Wert des Argumentes darstellt. X wird zu einem Integer-Wert gerundet, bevor OCT¤(X) berechnet wird (siehe die HEX¤-Funktion fuer Hexadizimalkonvertierungen).

Beispiel

```
PRINT OCT¤(24)
30
```

HEX¤(X)

Die Funktion liefert eine Zeichenkette, die den hexadezimalen Wert des Argumentes darstellt. X wird zu einem Integer-Wert gerundet, bevor HEX¤(X) ausgefuehrt wird (siehe die OCT¤-Funktion fuer Oktalkonvertierungen).

Beispiel

```
PRINT HEX¤(10)
A
```

STR¤(X)

Die Funktion STR¤ konvertiert einen numerischen Wert in dessen Zeichenkettendarstellung. Das erste Zeichen der Zeichenkette enthaelt das Vorzeichen. Anstelle eines positiven Vorzeichens erscheint ein Leerzeichen.

Beispiel

```
10  REM AUSGABE EINER ZAHL MIT KOMMA ANSTELLE DES
11  REM DEZIMALPUNKTES
20  INPUT X
30  X¤=STR¤(X)
40  P%=INSTR(X¤,".")
50  IF P%<>0
       THEN X¤=LEFT¤(X¤,P%-1)+","+RIGHT¤(X¤,LEN(X¤)-P%)
60  PRINT X,X¤

RUN
---
?  66
66          66
```

```
RUN
---
? 184.51
184.51       184,51
```

CHR¤(I)

Die Funktion liefert eine Zeichenkette, bestehend aus einem Zeichen, das den KOI-7-Code I besitzt. CHR¤ wird gewoehnlich zum Senden spezieller Zeichen an ein Terminal verwendet. Zum Beispiel kann das fuer Bildschirmsteuerfolgen benoetigte <ESC> durch CHR¤(27) gebildet werden.

Beispiel

```
PRINT CHR¤(66)
B
```

7.6. Standardfunktionen fuer die Zeichenkettenverarbeitung

LEN(X¤)

Die Funktion liefert die Zahl der Zeichen in der Zeichenkette X¤. Nichtdruckbare Zeichen und Leerzeichen werden mitgezaehlt.

Beispiel

```
10  X¤="BASIC INTERPRETER"
20  PRINT LEN(X¤)

RUN
---
 17
```

LEFT¤(X¤,I)

Die Funktion liefert eine Zeichenkette, die aus den linken I Zeichen von X¤ besteht.
Es muss 0<=I<=255 gelten. Gilt I>=LEN(X¤), so wird die gesamte Zeichenkette X¤ als Ergebnis geliefert. Gilt I=0, so ist das Ergebnis eine leere Zeichenkette (siehe auch die Funktionen MID¤ und RIGHT¤).

Beispiel

```
10  A¤="BASIC-INTERPRETER"
20  B¤=LEFT¤(A¤,5)
30  PRINT B¤

RUN
---
BASIC
```

RIGHT¤(X¤,I)

Die Funktion liefert eine Zeichenkette aus I Zeichen, die aus den am weitesten rechts stehenden Zeichen von X¤ gebildet wird. Wenn I>=LEN(X¤) gilt, so ist das Resultat mit X¤ identisch. Fuer I=0 wird eine leere Zeichenkette geliefert (siehe auch die Funktionen MID¤ und LEFT¤).

Beispiel

```
5 REM DRUCK EINER ZAHL OHNE FUEHRENDES LEERZEICHEN
10 A¤=STR¤(12.52)
20 B¤=RIGHT¤(A¤,LEN(A¤)-1
30 PRINT A¤
40 PRINT B¤

RUN
 12.52
12.52
```

MID¤(X¤,I[,J])

Die Funktion liefert einen string-Wert mit J Zeichen aus X¤, beginnend mit dem I-ten Zeichen in X¤. Es muss 1<=I<=255 und 0<=J<=255 gelten. Ist J nicht angegeben oder besteht die Zeichenkette, die mit dem I-ten Zeichen in X¤ beginnt, aus weniger als J Zeichen, so wird die gesamte Zeichenkette, die in X¤ mit dem I-ten Zeichen beginnt, als Resultat geliefert. Gilt I>LEN(X¤) oder J=0, so liefert MID¤ eine leere Zeichenkette (siehe auch die Funktionen LEFT¤ und RIGHT¤). Wenn I und J nicht im geforderten Bereich liegen, wird ein Fehler gemeldet.

Beispiel

```
10  A¤="GUTEN "
20  B¤="MORGEN TAG ABEND"
30  PRINT A¤;MID¤(B¤,8,3)
40  PRINT A¤+MID¤(B¤,12)
50  PRINT A¤+MID¤(B¤,,6)

RUN
GUTEN TAG
GUTEN ABEND
GUTEN MORGEN
```

INSTR([I,]X¤,Y¤)

Die Funktion sucht das erste Auftreten der Zeichenkette Y¤ in der Zeichenkette X¤ und liefert die Position, an der diese Zeichenkette auftritt.
Der wahlweise Parameter I bestimmt die Position, an der mit der Suche begonnen wird. Es muss 1<=I<=255 gelten. Ist I nicht angegeben, so wird 1 angenommen. Wenn I>LEN(X¤) gilt, X¤ leer ist oder Y¤ nicht gefunden wird, liefert INSTR den Wert 0. Wenn Y¤ leer ist, liefert INSTR entweder

I oder 1. X¤ und Y¤ koennen String-Variable, String-Ausdruecke oder string-Konstanten sein. Wenn I nicht im geforderten Intervall von 1 bis 255 liegt, wird ein Fehler gemeldet.

Beispiel

```
10  X¤="SCHLUESSEL"
20  Y¤="L"
30  PRINT INSTR(X¤,Y¤);INSTR(5,X¤,Y¤)

RUN
 4  10
```

SPACE¤(X)

Die Funktion liefert einen String-Wert, der aus X Leerzeichen besteht. Das Argument X wird in einen Integer-Wert gerundet und muss im Bereich von 0 bis 255 liegen (siehe auch die SPC-Funktion).

Beispiel

```
10  FOR I=1 TO 5
20  X¤=SPACE¤(I)
30  PRINT X¤;I
40  NEXT I

RUN
 1
  2
   3
    4
     5
```

STRING¤(I,J)

STRING¤(I,X¤)

Die Funktion liefert einen String-Wert aus I Zeichen, die alle entweder den KOI-7-Code J haben oder dem ersten Zeichen von X¤ entsprechen.

Beispiel

```
10  X¤=STRING¤(10,45)
20  PRINT X¤ "BERICHT" X¤

RUN
----------BERICHT----------
```

INPUT¤(X) 0<X<=255

Die Funktion liefert eine Zeichenkette von X Zeichen, die vom Terminal gelesen werden.
Die eingegebenen Zeichen werden nicht auf dem Terminal ausgeschrieben (kein Echo).
Es koennen alle Control-Zeichen eingegeben werden, ausser CTRL/C, das zur Unterbrechung der Ausfuehrung der INPUT¤-Funktion verwendet wird.

Beispiel

```
100 PRINT "WEITER ?--EINGABE(J/N):"
110  X¤=INPUT¤(1)
120  IF X¤="J" GOTO 2000
130  IF X¤="N" GOTO 1000 ELSE 100
 .
 .
```

INKEY¤

Es wird eine Zeichenkette, bestehend aus einem vom Terminal gelesenen Zeichen geliefert. Falls kein Zeichen am Terminal anliegt, wird eine leere Zeichenkette bereitgestellt. Die so eingelesenen Zeichen erscheinen nicht auf den Terminal (kein Echo). Es koennen alle Zeichen (mit Ausnahme von CTRL/C) eingelesen werden. CTRL/C beendet die Programmabarbeitung.

Beispiel

```
10 REM SUBROUTINE ZUR PASSWORTEINGABE
20 PASSWORD¤=""
30 PRINT "ENTER PASSWORD"
40 A¤=INKEY¤
50 IF A¤= "" THEN GOTO 40
60 IF ASC(A¤)=13 THEN RETURN
70 PASSWORD¤=PASSWORD¤+A¤
80 GOTO 40
```

7.7. Servicefunktionen

RND[(X)]

Die Funktion liefert eine Zufallszahl zwischen 0 und 1. Bei jedem Programmstart durch ein RUN-Kommando wird der Zufallsgenerator initialisiert. Nach einer Initialisierung liefert er stets die gleiche Folge von Zufallszahlen, falls nicht durch eine RANDOMIZE-Anweisung (s. Abschn. 6.3.4.) die Erzeugung einer anderen Folge veranlasst wird. RND ohne Argument oder mit X>0 liefert die naechste Zufallszahl in der Folge. RND mit X=0 wiederholt die zuletzt generierte Zufallszahl. RND mit X<0 initialisiert zunaechst den Zufallsgenerator (jedoch anders als beim RUN-Kommando) und liefert dann die erste Zufallszahl dieser Folge.

Beispiel

```
10 FOR I = 1 TO 6
20 LPRINT RND
30 NEXT

RUN
 .1213501
 .651861
 .8688611
 .7297625
 .798853
 7.369805E-02
```

FRE(X)

FRE(X¤)

Die Funktion liefert den Umfang des von BASIC noch nicht belegten Speichers in Byte.
Ist das Argument von FRE eine leere Zeichenkette, so wird zunaechst eine Garbage Collection durchgefuehrt, bevor der Speicherumfang bestimmt wird. Zu beachten ist, dass eine Garbage Collection 60 bis 90 Sekunden dauern kann. BASIC fuehrt eine Garbage Collection von sich aus erst dann aus, wenn kein freier Speicher mehr vorhanden ist. Deshalb sollte FRE("") periodisch angewandt werden, um kuerzere Laufzeiten fuer die Garbage Collecion zu erzielen.

Beispiel

```
PRINT FRE("")
 62367
```

7.8. Funktionen zur Drucker-/Bildschirmsteuerung

TAB(I)

Die Funktion stellt auf dem Terminal durch Einfuegen von Leerzeichen die Position I ein.
Ist die aktuelle Druckposition bereits groesser als I, so wird die Position I auf der naechsten Zeile eingestellt. Die am weitesten links stehende Position ist 1, die am weitesten rechts stehende entspricht der Zeilenweite minus 1. Es muss 1<=I<=255 gelten. TAB darf nur im Zusammenhang mit einer PRINT oder LPRINT-Anweisung verwendet werden.

Beispiel

```
10  PRINT "NAME" TAB(20) "VORNAME"
20  PRINT STRING¤(26,"*")
30  READ N¤,V¤
40  PRINT N¤ TAB(20) V¤
50  DATA MUELLER,MAX

RUN
---
NAME                 VORNAME
**************************
MUELLER              MAX
```

SPC(I)

Die Funktion gibt I Leerzeichen auf dem Terminal bzw. dem Drucker aus. SPC kann nur im Zusammenhang mit einer PRINT- oder LPRINT-Anweisung verwendet werden. Es muss 0<=I<=255 gelten (siehe auch die SPACE¤-Funktion).

Beispiel

```
PRINT "A" SPC(5) "B"
A      B
```

LPOS(X)

Die Funktion liefert die aktuelle Position innerhalb des Ausgabepuffers fuer den Drucker.
Diese Position muss nicht mit der physischen Position des Druckerkopfes uebereinstimmen. X ist ein Scheinargument.

Beispiel

```
100  IF LPOS(X)>60 THEN LPRINT CHR¤(13)
```

POS(X)

Die Funktion liefert die aktuelle Kursorposition des Terminals. Die am weitesten links stehende Position ist 1. X ist ein Scheinargument (siehe auch die Funktion LPOS).

Beispiel

```
IF POS(X)>60 THEN PRINT CHR¤(13)
```

8. Anweisungen und Funktionen fuer die Dateiarbeit

8.1. Allgemeines

Es gibt zwei Arten von Dateien auf Disketten, die durch BASIC-Programme erzeugt und verarbeitet werden koennen:
- sequentielle Dateien und
- Direktzugriffsdateien.

Sequentielle Dateien

Bei sequentiellen Dateien werden die auszugebenden Daten in der Reihenfolge ihres Auftretens in Textform zur Ausgabedatei uebertragen. Die Eingabe ist nur in der gleichen Reihenfolge moeglich. Sequentielle Dateien werden angewendet, wenn die Ein-oder Ausgabe auf ein Geraet erfolgt, das nur sequentielle Arbeit zulaesst oder wenn die Datei anschliessend durch den Texteditor, Druckprogramme usw. verarbeitet werden soll. Zur Zwischenspeicherung von Daten, die durch andere BASIC-Programme verarbeitet werden sollen, koennen guenstiger Direktzugriffsdateien verwendet werden.

Direktzugriffsdateien

Die Erzeugung und Verarbeitung von Direktzugriffsdateien erfordert mehr Programmschritte als bei sequentiellen Dateien, jedoch weist die Verwendung von Direktzugriffsdateien einige Vorteile auf. Ein Vorteil ist, dass Direktzugriffsdateien weniger Platz auf der Diskette benoetigen, weil die Daten im gepackten Binaerformat gespeichert werden (bei sequentiellen Dateien werden die Daten als Folgen von ASCII-Zeichen gespeichert).
Der wichtigste Vorteil der Direktzugriffsdateien ist, dass zu den Daten direkt zugegriffen werden kann, d.h. ohne vorher alle vorangehenden Daten lesen zu muessen, wie es bei sequentiellen Dateien der Fall ist. Dies ist moeglich, weil die Daten in sogenannten Datensaetzen gespeichert werden und jeder Datensatz numeriert ist.

Die Verarbeitung einer Datei setzt voraus, dass sie eroeffnet wird. Nur im eroeffneten Zustand kann auf die Daten einer Datei zugegriffen werden. Ist die Verarbeitung beendet, so muss die Datei geschlossen werden. Zum Eroeffnen und Schliessen von Dateien existieren entsprechende Anweisungen, die im naechsten Abschnitt beschrieben werden. Anschliessend werden die Anweisungen und Funktionen fuer den Zugriff auf sequentielle und Direktzugriffsdateien beschrieben. Zahlreiche Beispiele sollen das Verstehen erleichtern.

8.2. Eroeffnen und Schliessen von Dateien

8.2.1. OPEN

Syntax

```
OPEN <dateimodus>,[#]<dateinummer>,<dateispezifikation>
     [,<satzlaenge>]
```

Semantik

<dateimodus> ist ein Zeichenkettenausdruck, dessen erstes Zeichen eines der folgenden sein muss:
O - spezifiziert den sequentiellen Ausgabemodus
I - spezifiziert den sequentiellen Eingabemodus
R - spezifiziert den direkten Eingabe/Ausgabemodus

<dateinummer> ist ein Integer-Ausdruck, dessen Wert in dem durch die F-Option festgelegten Wertebereich liegen muss. Wird die F-Option beim Start des BASIC-Interpreters nicht angegeben, so kann die Dateinummer einen Wert zwischen 1 und 3 annehmen.

<dateispezifikation> gibt den Namen der zu oeffnenden Datei an.

<satzlaenge> ist ein Integer-Ausdruck, dessen Wert im Bereich 1 bis 32767 liegen kann und die Satzlaenge fuer Direktzugriffsdateien festlegt (siehe FIELD-Anweisung). Fehlt die Satzlaenge, so werden 128 Byte angenommen. Die hier angegebene Satzlaenge darf nicht die beim Start des BASIC-Interpreters festgelegte Satzlaenge durch die S-Option ueberschreiten.

Die OPEN-Anweisung ordnet der Datei einen Puffer fuer die E/A zu und bestimmt den Zugriffsmodus. Ist einmal eine Datei eroeffnet, so kann diese ueber die Dateinummer zur E/A genutzt werden. Eine OPEN-Anweisung muss folgenden Anweisungen vorausgehen:

PRINT#, WRITE#, PRINT# USUNG, INPUT#, LINE INPUT#, GET, PUT

GET- und PUT-Anweisungen koennen nur bei Direktzugriffsdateien verwendet werden.
Ist kein Geraetename angegeben, so wird das augenblicklich aktive Diskettengeraet angenommen.
Die maximale Anzahl von Dateien, die waehrend eines Programmlaufs gleichzeitig geoeffnet werden koennen, wird durch Angabe der F-Option beim Start des BASIC-Interpreters bestimmt. Standardmaessig koennen gleichzeitig 3 Dateien geoeffnet werden.
Existiert die spezifizierte Datei nicht, so wird bei INPUT ein "FILE NOT FOUND"-Fehler ausgewiesen und bei OUTPUT eine neue Datei angelegt. Pro Dateinummer darf jeweils nur eine Datei geoeffnet werden.

8.2.2. CLOSE

Syntax

CLOSE[[#]<dateinummer>[,[#]<dateinummer>...]]

Semantik

CLOSE schliesst die Dateien, die unter den angegebenen Dateinummern mit OPEN eroeffnet wurden.
Wird keine Dateinummer angegeben, so werden alle Dateien geschlossen. Nach Ausfuehrung der CLOSE-Anweisung besteht keine Verbindung mehr zwischen der Dateinummer und der darunter eroeffneten Datei. Die geschlossene Datei kann dann (auch unter anderer Nummer) von neuem eroeffnet werden und die alte Dateinummer kann fuer das Eroeffnen einer anderen Datei verwendet werden.
Eine CLOSE-Anweisung fuer sequentielle Ausgabedateien bewirkt die Ausgabe eventuell noch im Speicher stehender Ausgabedaten.
Die END- und die NEW-Anweisung fuehren ebenfalls zum Schliessen aller Dateien, waehrend bei der Programmbeendigung durch STOP die Dateien eroeffnet bleiben.

Beispiel

```
CLOSE#1,2
CLOSE
```

8.3. E/A mit sequentiellen Dateien

Die folgenden Anweisungen und Funktionen sind fuer die Arbeit mit sequentiellen Dateien vorgesehen.

OPEN	PRINT#	INPUT#	WRITE#
	PRINT# USING	LINE INPUT#	
CLOSE	EOF	LOC	LOF

8.3.1. PRINT# und PRINT# USING

Syntax

PRINT#<dateinummer>,[USING<zeichenkettenausdruck>;]
<liste von ausdruecken>

Semantik

Unter der spezifizierten Dateinummer muss eine eroeffnete Ausgabedatei vorliegen.
Der Zeichenkettenausdruck nach USING stellt eine Aufbereitungsvorschrift fuer die auszugebenden Werte dar (s. Abschn. 6.4.4.).
Die in der Ausdrucksliste enthaltenen numerischen oder Zeichenkettenausdruecke werden in die Ausgabedatei geschrieben.
Die Daten werden so, wie sie bei einer entsprechenden PRINT-Anweisung auf dem Bildschirm erscheinen wuerden, in die Datei geschrieben. Deshalb sollten numerische Ausdruecke in der Ausdrucksliste durch Semikolon getrennt werden (Komma als Trenner bewirkt die Einteilung der Ausgabezeile in Druckzonen durch Einfuegen zusaetzlicher Leerzeichen).
Zeichenkettenausdruecke muessen in der Ausgabeliste durch Semikolon getrennt werden. Sollen einmal ausgegebene Zeichenkettenausdruecke spaeter wieder einzeln eingelesen werden, so muessen diese ordentlich voneinander getrennt in die Datei gespeichert werden. Um dies zu erreichen, muessen explizit Trenner in die Liste der auszugebenden Ausdruecke eingefuegt werden.

Beispiele

A¤="NAME"
B¤="VORNAME"

Die Anweisung PRINT#1,A¤;B¤ bewirkt die Ausgabe der Zeichenfolge

NAMEVORNAME

in die Datei. Die Zeichenketten "NAME" und "VORNAME" koennen nicht mehr voneinander getrennt gelesen werden.
Um das zu aendern, wird ein Komma zwischen die Zeichenketten geschoben.
Durch PRINT#1,A¤;",";B¤ wird die Zeichenfolge

NAME,VORNAME

ausgegeben.
Diese Zeichenketten koennen mit der Anweisung INPUT#1,A¤,B¤ wieder eingelesen werden.

Falls die auszugebenden Zeichenketten selbst Trennzeichen enthalten, die fuer die INPUT#-Anweisung signifikant sind (Komma, Leerzeichen, carriage return, line feed), so ist es ratsam, die Zeichenkettenausdruecke explizit in Anfuehrungsstriche einzuschliessen. Dazu wird die Funktion CHR¤(34) verwendet.

```
A¤="NAME,VORNAME"
B¤="WOHNORT,STRASSE"

PRINT#1,CHR¤(34);A¤;CHR¤(34);CHR¤(34);B¤;CHR¤(34)
```

bewirkt die Ausgabe der Zeichenfolge:

```
"NAME,VORNAME""WOHNORT,STRASSE"
```

Die beiden Zeichenketten koennen durch die Anweisung

```
INPUT#1,A¤,B¤
```

gelesen werden, wobei A¤ den Wert "NAME,VORNAME" und B¤ den Wert "WOHNORT,STRASSE" erhalten.

Bei der formatierten Ausgabe mit der USING-Option koennen die erforderlichen Trennzeichen in der Aufbereitungsvorschrift angegeben werden.

```
J=20.3
K=4.46

PRINT#1,USING"**#.##,";J;K      bewirkt die Ausgabe von

*20.30,**4.46,
```

8.3.2. WRITE#

Syntax

```
WRITE#<dateinummer>,<liste von ausdruecken>
```

Semantik

Mit Hilfe einer WRITE#-Anweisung erfolgt die Ausgabe von Datensaetzen in eine sequentielle Datei.
Unter der angegebenen Dateinummer muss eine eroeffnete Ausgabedatei vorliegen (s. Abschn. 8.2.1.). Die Liste kann numerische Ausdruecke oder Zeichenkettenausdruecke, die durch Komma getrennt sind, enthalten.
Die Werte der aufgefuehrten Ausdruecke werden in die sequentielle Datei geschrieben. Im Unterschied zur PRINT-Anweisung werden zwischen den einzelnen Datenelementen Kommas ausgegeben, und hinter dem letzten Datenelement der Liste wird automatisch ein "carriage return"/"line feed" gespeichert. Zeichenketten werden in Anfuehrungsstriche eingeschlossen.
Bei der WRITE#-Anweisung brauchen nicht explizit Trenner zwischen den Datenelementen in der Liste eingefuegt werden. Die einzelnen Datenelemente koennen auch so durch eine INPUT#-Anweisung wieder gelesen werden.

Beispiel

A¤="NAME,VORNAME"
B¤="WOHNORT"

Die Anweisung WRITE#1,A¤,B¤ schreibt folgenden Satz zur Datei:
"NAME,VORNAME","WOHNORT"
Durch die Anweisung INPUT#1,A¤,B¤ koennen diese Werte wieder eingelesen werden, so dass "NAME,VORNAME" der Variablen A¤ und "WOHNORT" der Variablen B¤ zugewiesen werden.

8.3.3. INPUT#

Syntax

INPUT#<dateinummer>,<variablenliste>

Semantik

Die INPUT#-Anweisung wird zur Dateneingabe von einer sequentiellen Plattendatei verwendet.
Unter der spezifizierten Dateinummer muss eine eroeffnete Eingabedatei vorliegen (s. Abschn. 8.2.1.).
Die Variablenliste enthaelt die Namen aller Variablen, denen bei der Eingabe von der Datei Werte zuzuordnen sind. Durch die Variablennamen wird festgelegt, welcher Datentyp von den Eingabedaten erwartet wird.

Notationsvorschrift fuer die Eingabedaten:

- Numerische Werte
  Fuehrende Leerzeichen, "carriage return" und "line feed" werden bei der Eingabe ignoriert. Das erste Zeichen, welches keinem Leerzeichen, "carriage return" oder "line feed" entspricht, wird als Anfang der numerischen Dateneinheit angenommen. Abgeschlossen wird die Dateneinheit durch ein Leerzeichen, "carriage return", "line feed" oder Komma.

- Zeichenkettenwerte
  Fuehrende Leerzeichen, "carriage return" und "line feed" werden ignoriert. Das erste Zeichen, welches keinem Leerzeichen, "carriage return" oder "line feed" entspricht, wird als Anfang der Zeichenkette gewertet.
  Werden als erstes Zeichen Anfuehrungsstriche erkannt, so besteht die Zeichenkette selbst aus allen Zeichen, die zwischen den einleitenden und abschliessenden Anfuehrungsstrichen gelesen werden. Folglich kann eine durch Anfuehrungsstriche begrenzte Zeichenkette selbst keine Anfuehrungsstriche enthalten.
  Eine Zeichenkette, die nicht durch Anfuehrungsstriche eingeleitet wird, wird durch Auftreten eines Kommas, "carriage return", "line feed" oder wenn 255 Zeichen gelesen worden sind, abgeschlossen.
  Beim Erreichen des Dateiendes (EOF), wird die gelesene Dateneinheit abgeschlossen.

8.3.4. LINE INPUT#

Syntax

LINE INPUT#<dateinummer>,<zeichenkettenvariable>

Semantik

Die LINE-INPUT#-Anweisung dient der Eingabe eines Datensatzes von einer sequentiellen Datei.
Unter der in der Anweisung spezifizierten Dateinummer muss eine eroeffnete Eingabedatei vorliegen.
Der von der Datei gelesene Datensatz wird als Zeichenkette interpretiert und der angegebenen Zeichenkettenvariable als Wert zugewiesen.
Ein Datensatz wird durch "carriage return" abgeschlossen, d.h. es werden alle Zeichen bis zum Erkennen eines "carriage return" aus der Datei gelesen.
Nach dem Lesen eines Datensatzes wird sofort auf den Anfang des folgenden Datensatzes positioniert.

Hinweis

Die LINE-INPUT#-Anweisung ist besonders geeignet, wenn jede Datenzeile einer Datei in mehrere Felder unterteilt ist, die getrennt verarbeitet werden sollen (MID¤).

Beispiel

```
10  OPEN "O",1,"LISTE"
20  LINE INPUT "TEILNEHMERINFORMATION?";Z¤
30  PRINT#1,Z¤
40  CLOSE 1
50  OPEN "I",1,"LISTE"
60  LINE INPUT#1,Z¤
70  PRINT Z¤
80  CLOSE 1

RUN
---
TEILNEHMERINFORMATION? ANTON MUELLER  8010 DRESDEN
                       --------------------------
ANTON MUELLER  8010 DRESDEN
```

8.3.5. Erstellen, Verarbeiten und Erweitern von sequentiellen Dateien

Folgende Programmschritte sind erforderlich, um eine sequentielle Datei zu erzeugen und die Daten zu lesen:

Programmschritt	Beispiel
- Eroeffnen der Datei im O-Modus	OPEN "O",#1,"DATEI"
- Schreiben von Daten in die Datei unter Verwendung der PRINT#-, PRINT# USING- oder WRITE#-Anweisung	PRINT#1,A¤;B¤;C¤
- Schliessen der Datei und	CLOSE#1
Eroeffnen im I-Modus	OPEN "I",#1,"DATEI"
- Lesen von Daten aus der Datei unter Verwendung der INPUT#-, oder LINE-INPUT#-Anweisung	INPUT#1,A¤;B¤;C¤ LINE INPUT#1,ZEILE¤

Hinweis

Die LOC-Funktion liefert die Anzahl der Sektoren (128 Byte-Bloecke), die in die sequentielle Datei geschrieben bzw. aus dieser gelesen wurden. Die EOF-Funktion dient zur Erkennung des Dateiendes.

Beispiel

```
IF EOF(1) THEN END
```

Anfuegen von Daten an eine sequentielle Datei

Sollen an eine auf einer Diskette vorhandene sequentielle Datei Daten angefuegt werden, so kann dies nur auf folgendem Wege geschehen:

- Eroeffnen von "DATEI" im I-Modus
- Eroeffnen einer weiteren Datei "KOPIE" im I-Modus
- Lesen aller Daten aus "DATEI" und Schreiben in "KOPIE"
- Schliessen und Loeschen (KILL) von "DATEI"
- Schreiben der neuen Daten nach "KOPIE"
- Schliessen von "KOPIE" und Umbezeichnen (NAME) in "DATEI"

Im Ergebnis dieser Schritte enthaelt "DATEI" alle bisherigen sowie alle neuen Daten.

Beispiele

Im folgenden Programmbeispiel wird eine sequentielle Datei mit dem Namen "DATE" erzeugt. Es werden Informationen, die ueber Terminal eingegeben werden, verarbeitet.

```
10  OPEN "O",#1,"DATE"
20  INPUT "NAME";N¤
30  IF N¤="*" THEN 80
40  INPUT "GEBURTSDATUM";D¤
50  INPUT "ANSCHRIFT";A¤
60  PRINT#1,N¤;",";D¤;",";A¤
70  GOTO 20
80  CLOSE#1

RUN
---
NAME?  MUELLER,MAX
       -----------
GEBURTSDATUM?  24.03.53
               --------
ANSCHRIFT?  9999 ADORF, LANGE GASSE 14
            --------------------------
NAME?
...
...
```

Im naechsten Beispiel wird die im obigen Programm erstellte Datei "DATE" verarbeitet. Es werden die Namen aller Personen, die 1953 geboren wurden, ausgegeben.

```
10  OPEN "I",#1,"DATE"
20  IF EOF(1) THEN 60
30  INPUT#1,N¤,D¤,A¤
40  IF RIGHT¤(D¤,2)="53" THEN PRINT N¤
50  GOTO 20
60  CLOSE#1

RUN
---
MUELLER,MAX
.
.
```

Das Programm liest sequentiell alle Daten der Datei. Bevor eine Eingabeanweisung erfolgt (Zeile 30), wird zunaechst ueberprueft, ob das Ende der Datei erreicht wurde oder nicht (Zeile 20). Wird diese Ueberpruefung nicht vorgenommen, so wird bei Erreichen des Dateiendes ein Fehler gemeldet.
Ein Programm, das eine sequentielle Datei erzeugt, kann auch formatierte Daten mit Hilfe der PRINT#-USING-Anweisung auf die Diskette schreiben. Zum Beispiel kann die Anweisung

```
PRINT#1,USING "####.##,";A,B,C,D
```

verwendet werden, um numerische Daten ohne explizite Begrenzerzeichen zu schreiben. Das Komma am Ende der Formatzeichenreihe dient zur Separierung der einzelnen Zahlen.

Das folgende Programmbeispiel demonstriert die Technik fuer das Erweitern von sequentiellen Dateien.
Das Programm kann zur Erzeugung (falls Datei "DATE" noch nicht existiert) oder zur Erweiterung einer Datei mit dem Namen "DATE" verwendet werden.
Weiterhin wird hier die Verwendung der Anweisung LINE INPUT# zur Eingabe von Zeichenreihen mit Kommas gezeigt. Es sei daran erinnert, dass LINE INPUT# soviele Zeichen liest, bis ein "carriage return" gefunden wird oder bis 255 Zeichen gelesen wurden (Anfuehrungsstriche und Kommas werden mitgelesen).

```
10    ON ERROR GOTO 2000
20    OPEN "I",#1,"DATE"
30    REM FALLS DIE DATEI EXISTIERT, WIRD SIE KOPIERT
40    OPEN "O",#2,"KOPIE"
50    IF EOF(1) THEN 90
60    LINE INPUT#1,A¤
70    PRINT#2,A¤
80    GOTO 50
90    CLOSE#1
100   KILL "DATE"
110   REM ANFUEGEN NEUER DATEN
120   INPUT "NAME";N¤
130   IF N¤="*" THEN 180  '"*" beendet die Eingabeschleife
140   INPUT "GEBURTSDATUM";D¤
150   INPUT "ANSCHRIFT";A¤
160   WRITE#2,N¤,D¤,A¤
170   GOTO 120
180   CLOSE
190   NAME "KOPIE" AS "DATE"
200   END
2000  IF ERR=53 AND ERL=20 THEN OPEN "O",#2,"KOPIE":RESUME 120
2010  ON ERROR GOTO 0
```

Die Fehlerbehandlungsroutine untersucht, ob der Fehler "Datei nicht vorhanden" in der Zeile 20 auftritt. Ist dies der Fall, so werden die Anweisungen zum Kopieren der alten Daten uebergangen und somit eine neue Datei erstellt.

8.4. E/A mit Direktzugriffsdateien

Die folgenden Anweisungen und Funktionen sind fuer die Arbeit mit Direktzugriffsdateien vorgesehen:

OPEN	PUT	GET
CLOSE		
FIELD	LSET/RSET	
	MKD¤,MKS¤,MKI¤	CVI,CVS,CVD
LOC		

8.4.1. FIELD

Syntax

```
FIELD[#]<dateinummer>,<feldlaenge> AS <stringvariable>
                    [,<feldlaenge> AS <stringvariable>]....
```

Semantik

Mit Hilfe der FIELD-Anweisung werden String-Variablen im E/A-Puffer einer Direktzugriffsdatei angeordnet.
Die Saetze einer mit OPEN eroeffneten Direktzugriffsdatei werden entweder mit GET in den E/A-Puffer der Datei gelesen oder mit PUT von dort auf die Direktzugriffsdatei geschrieben. Um Daten aus diesem E/A-Puffer lesen zu koennen oder Daten dorthin zu bringen, werden Zeichenkettenvariablen im E/A-Puffer vereinbart.
Unter <feldlaenge> wird die Byteanzahl der Zeichenkette angegeben, auf die unter dem Namen <stringvariable> zugegriffen wird.
Die angegebenen Laenge-Name-Paare beschreiben den Satzaufbau der Direktzugriffsdatei.
Die Summe der angegebenen Feldlaengen darf die Satzlaenge der Direktzugriffsdatei nicht ueberschreiten, sonst wird der Fehler "field overflow" gemeldet.
Fuer ein und dieselbe Datei koennen mehrere FIELD-Anweisungen abgearbeitet werden, die alle gleichzeitig gelten. Kommt eine Variable jedoch mehrfach in FIELD-Anweisungen vor, so gilt die durch die letzte abgearbeitete FIELD-Anweisung hergestellte Zuordnung.
Um auch Integer- und Realdaten in Direktzugriffsdateien in ihrer Internform fuehren zu koennen, benutzt man die Konvertierungsfunktionen MKI¤, MKD¤, MKS¤ bzw. CVI, CVD, CVS.

Hinweis

Variablen, die in einer FIELD-Anweisung aufgefuehrt sind, duerfen nicht in einer INPUT-Anweisung und nicht auf der linken Seite einer Ergibtanweisung vorkommen, da sonst die Variable nicht mehr auf den E/A-Puffer der Datei verweist.
Fuer Zuweisungen an FIELD gebundene Variablen sind ausschliesslich in RSET- oder LSET-Anweisungen (s. Abschn. 8.4.2.) zu verwenden!

8.4.2. LSET und RSET

Syntax

```
LSET <zeichenkettenvariable>=<zeichenkettenausdruck>

RSET <zeichenkettenvariable>=<zeichenkettenausdruck>
```

Semantik

Mit LSET und RSET kann ein E/A-Puffer zur Vorbereitung einer PUT-Anweisung gefuellt werden.
Der Wert des Zeichenkettenausdruckes wird der Zeichenkettenvariablen, welche in einer FIELD-Anweisung vereinbart wurde, zugeordnet. Ist das durch die Zeichenkettenvariable reservierte Feld im E/A-Puffer groesser als benoetigt wird, so wird der Wert des Ausdruckes bei der LSET-Anweisung linksbuendig und bei der RSET-Anweisung rechtsbuendig eingetragen. Die freien Byte werden durch Leerzeichen aufgefuellt.
Ist der Wert des Zeichenkettenausdruckes zu lang, so werden rechts Zeichen abgeschnitten.
Numerische Werte muessen zu Zeichenkettenwerten konvertiert werden, bevor sie in den E/A-Puffer eingetragen werden. Dazu dienen die Funktionen MKI¤, MKS¤ UND MKD¤.

Beispiel

```
100  LSET B¤=X¤+Y¤
```

Hinweis

LSET und RSET koennen auch genutzt werden, um einer gegebenen Zeichenkettenvariablen einen Zeichenkettenwert linksbuendig bzw. rechtsbuendig zuzuordnen.

```
90   A¤=SPACE¤(10)
100  RSET A¤=N¤
```

Die Zeichenkette N¤ wird rechtsbuendig in das 10 Byte lange Zeichenkettenfeld A¤ eingetragen. Dies kann bei der formatierten Ausgabe verwendet werden.

8.4.3. PUT

Syntax

```
PUT[#]<dateinummer>[,<satznummer>]
```

Semantik

Mit der PUT-Anweisung wird der Datensatz, der sich im E/A-Puffer befindet, in die Direktzugriffsdatei geschrieben. Unter der angegebenen Dateinummer muss eine geoeffnete Direktzugriffsdatei vorliegen.
Ist keine Satznummer angegeben, so erfolgt die PUT-Anweisung mit der um 1 erhoehten Satznummer der vorangegangenen PUT-Anweisung.
Satznummern muessen im Wertebereich von 1 bis 32767 liegen.

Hinweis

Bevor eine PUT-Anweisung erfolgt, kann der E/A-Puffer auch durch eine PRINT#-, PRINT#-USING- oder WRITE#-Anweisung gefuellt werden. Im Falle von WRITE# wird der Puffer mit Leerzeichen bis zum "carriage return" aufgefuellt.

8.4.4. GET

Syntax

GET[#]<dateinummer>[,<satznummer>]

Semantik

Die GET-Anweisung wird zum Lesen eines Satzes von einer Direktzugriffsdatei in einen E/A-Puffer verwendet.
<dateinummer> ist die Nummer, unter welcher die Datei eroeffnet (OPEN) wurde. Fehlt die Angabe der Satznummer, so wird der naechste Datensatz (nach dem letzten GET) in den Puffer gelesen. Als groesste Satznummer ist 32767 zulaessig.

Hinweis

Nach einer GET-Anweisung koennen die eingegebenen Daten durch INPUT# oder LINE INPUT# aus dem Eingabepuffer gelesen werden.

8.4.5. MKI¤, MKS¤, MKD¤

Syntax

MKI¤(<integer-ausdruck>)

MKS¤(<einfach genauer ausdruck>)

MKD¤(<doppelt genauer ausdruck>)

Semantik

Es erfolgt eine Typanpassung von numerischen Werten in String-Werte.
Ein numerischer Wert, der in seiner internen Form im Puffer einer Direktzugriffsdatei abgespeichert werden soll, muss zunaechst durch eine Typanpassung als String-Wert betrachtet werden, um dann durch eine LSET- oder RSET-Anweisung abgespeichert werden zu koennen. Durch diese Typanpassung wird keine Aenderung des numerischen Wertes (Konvertierung) vorgenommen.
MKI¤ uebernimmt die Anpassung eines Integer-Wertes in einen 2 Zeichen langen String-Wert. MKS¤ uebernimmt die Anpassung eines einfach genauen Wertes in einen 4 Zeichen langen String-Wert.
MKD¤ uebernimmt die Anpassung eines doppelt genauen Wertes in einen 8 Zeichen langen String-Wert (s. Abschn. 8.4.6.).

Beispiel

```
90  AMT=(K+T)
100 FIELD#1,8 AS D¤,20 AS N¤
110 LSET D¤=MKS¤(AMT)
120 LSET N¤=A¤
130 PUT#1
.
.
.
```

8.4.6. CVI, CVS, CVD

Syntax

CVI(<zeichenkette aus 2 Zeichen>)

CVS(<zeichenkette aus 4 Zeichen>)

CVD(<zeichenkette aus 8 Zeichen>)

Semantik

Die Funktionen nehmen eine Typanpassung von String-Werten in numerische Werte vor. Numerische Werte (in Interndarstellung) werden beim Lesen von einer Direktzugriffsdatei den Zeichenkettenvariablen der FIELD-Anweisung als Werte zugewiesen. Ueber die Funktionen CVI, CVS und CVD koennen diese Werte ohne Aenderung wieder als numerische Werte verwendet werden.
CVI uebernimmt die Anpassung einer 2 Zeichen langen Zeichenkette in einen Integer-Wert. CVS uebernimmt die Anpassung einer 4 Zeichen langen Zeichenkette in einen einfach genauen Wert. CVD uebernimmt die Anpassung einer 8 Zeichen langen Zeichenkette in einen doppelt genauen Wert (s. Absch. 8.4.5.).

Beispiel

```
70  FIELD#1,8 AS D¤,20 AS N¤,...
80  GET#1
90  AMT=CVS(D¤)
100 A¤=N¤
.
.
.
```

8.4.7. Erstellen und Verarbeiten von Direktzugriffsdateien

Programmschritte zum Erstellen einer Direktzugriffsdatei

Programmschritt	Beispiel
- Eroeffnen der Datei im R-Modus. Es ist eine Satzlaenge von 28 Byte vorgesehen (Standard=128 Byte).	OPEN "R",#1,"DATEI",28
- Strukturierung des E/A-Puffers mit Hilfe einer FIELD-Anweisung	FIELD#1,26 AS Z¤,2 AS P¤
- Fuellen des E/A-Puffers mit Hilfe von LSET-/RSET-Anweisungen. Numerische Werte muessen in Zeichenkettenwerte verwandelt werden, dazu dienen MKI¤, MKD¤, MKS¤. (=Bereitstellung des Datensatzes)	RSET Z¤=X¤ LSET P¤=MKI¤(P%)
- Schreiben des Datensatzes in die Datei durch PUT-Anweisung.	PUT#1,SATZNUMMER%

Hinweis

Die in einer FIELD-Anweisung aufgefuehrten String-Variablen duerfen nicht in einer INPUT-Anweisung oder auf der linken Seite einer LET-Anweisung auftreten. In diesen Faellen wuerde der Zeiger fuer die Variablen in den String-Speicher verweisen und nicht mehr in den E/A-Puffer der Direktzugriffsdatei.

Folgende Schritte sind fuer die Verarbeitung von Direktzugriffsdateien notwendig:

- Eroeffnen der Datei im R-Modus	OPEN "R",#1,"PERSONAL",26
- Angabe einer FIELD-Anweisung zur Zuweisung von Speicherplatz im E/A-Puffer an die Variablen, die von der Datei gelesen werden sollen. *)	FIELD#1,2 AS P¤, 20 AS N¤,4 AS G¤
- Angabe einer GET-Anweisung zum Lesen des gewuenschten Datensatzes in den E/A-Puffer	GET#1,P%
- Die Daten des Puffers koennen nun verarbeitet werden. Bei numerischen Werten muss eine Typanpassung mit Hilfe der Funktionen CVI, CVS bzw. CVD vorgenommen werden.	PRINT CVI(P¤) PRINT N¤ PRINT CVS(G¤)

*) In einem Programm, in dem eine Direktzugriffsdatei sowohl gelesen als auch geschrieben wird, sind nur eine OPEN- und eine FIELD-Anweisung notwendig.

Hinweis

Mit Hilfe der LOC-Funktion kann fuer Direktzugriffsdateien die aktuelle Datensatznummer ermittelt werden. Als aktuelle Datensatznummer versteht man die in einer GET- oder PUT-Anweisung zuletzt benutzte Datensatznummer.
Zum Beispiel beendet die Anweisung

```
IF LOC(1)=50 THEN END
```

die Programmabarbeitung, wenn der Datensatz mit der Satznummer 50 verarbeitet worden ist.
Die EOF-Funktion kann fuer Direktzugriffsdateien nicht verwendet werden. Um festzustellen, wieviel Saetze eine Direktzugriffsdatei enthaelt, legt man den Satz 1 einer Direktzugriffsdatei als Kennsatz an. Ein Beispiel zur Arbeit mit Direktzugriffsdateien enthaelt die Anlage 4.

8.5. Funktionen

8.5.1. EOF

Syntax

EOF(<dateinummer>)

Semantik

Die EOF-Funktion ueberpruft, ob das Ende der durch die Dateinummer spezifizierten Eingabedatei erreicht ist.
Die Dateinummer wird in der Dateispezifikation der OPEN-Anweisung festgelegt. Die EOF-Funktion liefert bei der Anwendung auf eine sequentielle Eingabedatei den Wert -1 (true) oder 0 (false), je nachdem ob das Ende dieser Datei erreicht wurde oder nicht.
Die Anwendung der EOF-Funktion auf eine sequentielle Ausgabedatei liefert den Fehler "Bad file mode".
Die EOF-Funktion sollte verwendet werden, um vor dem Eingeben auf Dateiende zu pruefen. Ebenso kann die EOF-Funktion genutzt werden, um die Groesse einer Datei anhand eines Suchalgorithmus zu bestimmen. Eingaben nach Erreichen des Dateiendes fuehren zum Fehler "Input past end".

8.5.2. LOC

Syntax

LOC(<dateinummer>)

Semantik

Die LOC-Funktion liefert die aktuelle Position in der Datei. Bei Direktzugriffsdateien liefert LOC die Nummer des Satzes, der gerade gelesen bzw. geschrieben wurde.
Bei sequentiellen Dateien liefert LOC die Zahl der Sektoren (128-Byte-Bloecke), die gelesen bzw. geschrieben wurden, seitdem die

Datei eroeffnet wurde. Wenn eine sequentielle Datei im Eingabemodus eroeffnet wird, so wird sofort der erste Sektor gelesen und der Funktionswert von LOC ist 1.

8.5.3. LOF

Syntax

LOF (<dateinummer>)

Semantik

Die LOF-Funktion liefert die Anzahl der von der spezifizierten Diskettendatei belegten 128-Byte-Bloecke im Speicher. Die Datei muss geoeffnet sein.

8.5.4. INPUT¤

Syntax

INPUT¤(X[,[#]Y])

Semantik

Die INPUT¤-Funktion liefert eine Zeichenkette von X Zeichen, die entweder vom Terminal oder von der Datei mit der Nummer Y gelesen werden. Es gilt 0<X<=255.
Wird das Terminal zur Eingabe verwendet, so werden die eingegebenen Zeichen nicht auf dem Terminal ausgeschrieben (kein Echo). Es koennen alle Control-Zeichen eingegeben werden ausser CTRL/C, das zur Unterbrechung der Ausfuehrung der INPUT¤-Funktion verwendet wird.

Beispiel

```
 5 'AUSGABE DES INHALTES EINER SEQUENTIELLEN DATEI
 6 'IN HEXADEZIMALER FORM
10 OPEN "I",1,"DATA"
20 IF EOF(1) THEN 50
30 PRINT HEX¤(ASC(INPUT¤(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

8.5.5. VARPTR

Syntax

VARPTR(#<dateinummer>)

Semantik

Bei Anwendung auf sequentielle Dateien wird die Anfangsadresse des E/A-Puffers, der der Datei gemaess Dateinummer zugeordnet wurde, geliefert.
Bei Anwendung auf Direktzugriffsdateien wird die Adresse des FIELD-Puffers, der der Datei gemaess Dateinummer zugeordnet wurde, geliefert.

Beispiel

```
VARPTR(#1)
```

9. Rechnerspezifische Ausdrucksmittel

9.1. Codeprogramme

BASIC bietet die Moeglichkeit, ueber die USR-Funktion und die CALL-Anweisung Codeprogramme (Unterprogramme in der Assemblersprache) aufzurufen.
Die USR-Funktion erlaubt es, ein Codeprogramm in der gleichen Art und Weise zu rufen wie eine Standardfunktion. Der bessere Weg zum Aufruf von Codeprogrammen ist jedoch die CALL-Anweisung. Sie ist mit mehreren Sprachen kompatibel (im Gegensatz zur USR-Funktion), liefert einen besser lesbaren Quellcode und kann mehrere Argumente erhalten.

Speicherplatzzuweisung

Bevor ein Codeprogramm geladen werden kann, muss Speicherplatz fuer dieses reserviert werden.
Codeprogramme werden in das Datensegment (ueber DS adressiert) des BASIC-Interpreters geladen. Waehrend der Initialisierung von BASIC wird mit Hilfe der M-Option die Anfangsadresse des Codeprogrammes innerhalb des Datensegmentes festgelegt (siehe BASIC-Initialisierung). Mit einer DEF-SEG-Anweisung wird die Codesegmentadresse (CS) eingestellt, die bei der Ausfuehrung von Unterprogrammrufen und PEEK- und POKE-Anweisungen zur Adressierung verwendet wird.
Das Laden von Codeprogrammen in den reservierten Speicherbereich kann mit Hilfe der POKE-Anweisung erfolgen. Das Codeprogramm muss vorher assembliert und verbunden, aber nicht geladen werden. Zum Laden einer derart vorbereiteten Datei sind folgende Richtlinien zu beachten:
- Die Codeprogramme duerfen keine langen Referenzen enthalten.
- Die ersten 128 Byte des vom Linker erzeugten Lademoduls (=Dateiheader) muessen uebersprungen werden.

Das folgende Beispiel demonstriert das Laden von Codeprogrammen.

Laden des assemblierten und verbundenen Codeprogramms BSP.EXE in die hintere Haelfte des BASIC-internen Datensegmentes.

```
>BASIC /M:&H8000
```

Initialisierung
Nur die ersten 32k des Datensegmentes stehen BASIC zur Verfuegung, der restliche Speicher wird fuer Codeprogramme reserviert.

```
10 REM BASIC-PROGRAMM ZUM LADEN VON CODEPROGRAMMEN
20 OPEN "I",#1,"BSP.EXE"          'Oeffnen Eingabedatei
30 X¤=INPUT¤(128,#1)              'Uebergehen Dateiheader
40 DEF SEG                        'Segmentadresse=Datensegmentadresse
                                  'des  BASIC-Interpreters
50 I%=&H8000                      'Zaehler:= Offset bezueglich Daten-
                                  'segment  (Anfangsadresse des Code-
                                  'programms im Datensegment)
60 IF EOF(1) THEN END
70 POKE I%,ASC(INPUT¤(1,#1))      'Zeichenweises Laden der Datei
80 I%=I%+1
90 GOTO 60
```

Ist dem Nutzer die Datensegmentadresse des BASIC-Interpreters bekannt, so koennen die Programmzeilen 40 und 50 auch wie folgt aussehen:

```
40 DEF SEG=&H3800                 'Codesegment wird direkt auf die
50 I%=0                           'Anfangsadresse des Codeprogramms
                                  'gestellt.
```

(Hier sei angenommen, dass das Datensegment bei der absoluten Adresse H30000 beginnt.)

9.1.1. CALL-Anweisung

Syntax

CALL <variablenname>[(<argumentliste>)]

Semantik

Die CALL-Anweisung wird zum Aufruf externer Unterprogramme benutzt.
Die durch den Variablennamen bezeichnete Variable muss den Segmentoffset des Eintrittspunktes des aufzurufenden Unterprogramms enthalten. Die Argumentliste enthaelt durch Komma getrennte Variablen oder Konstanten, die dem Unterprogramm als Parameter uebergeben werden, bzw. welche die Resultatwerte nach Abarbeitung des Unterprogramms enthalten.
Die Adresse, zu der die CALL-Anweisung verzweigt, berechnet sich aus dem in der Variablen enthaltenen Segmentoffset und der Segmentadresse, welche durch eine vorausgegangene DEF-SEG-Anweisung festgelegt worden ist.
Die Ausfuehrung einer CALL-Anweisung geschieht in folgenden Schritten:

- Fuer jeden Parameter in der Argumentliste wird der 2-Byte-Offset des Parameters im Datensegment (DS) gekellert.
- Die Ruecksprungadresse (bestehend aus Codesegmentadresse CS und Offset IP) wird gekellert.
- Die Steuerung wird dem Codeprogramm durch einen "langen" Ruf uebergeben. Dabei wird als Codesegmentadresse die Adresse verwendet, die in der letzten DEF-SEG-Anweisung angegeben wurde, und als Offset der Inhalt der nach CALL angegebenen Variablen.

Diese Aktionen werden in den zwei folgenden Diagrammen demonstriert. Das erste Diagramm zeigt den Kellerzustand zur Zeit der Abarbeitung der CALL-Anweisung und das zweite den Kellerzustand waehrend der Abarbeitung des gerufenen Codeprogramms.

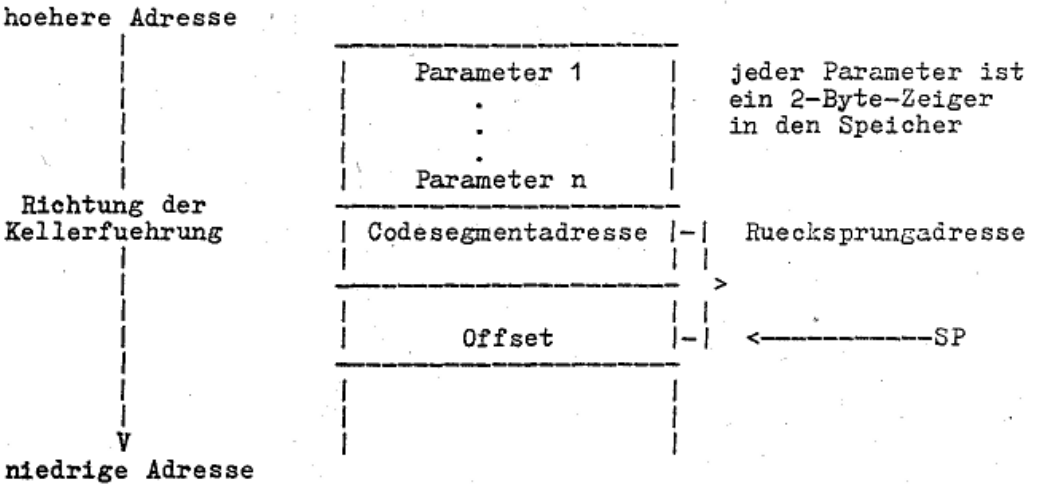


Bild 1: Kellerzustand zur Zeit der Abarbeitung der CALL-Anweisung

Nach dem Aufbau des Kellers gemaess dem obigen Bild wird die Steuerung dem Codeprogramm uebergeben. Auf Parameter kann zugegriffen werden, indem der Inhalt des Kellerzeigers SP in das Register BP uebertragen wird und dazu ein positiver Offset addiert wird.

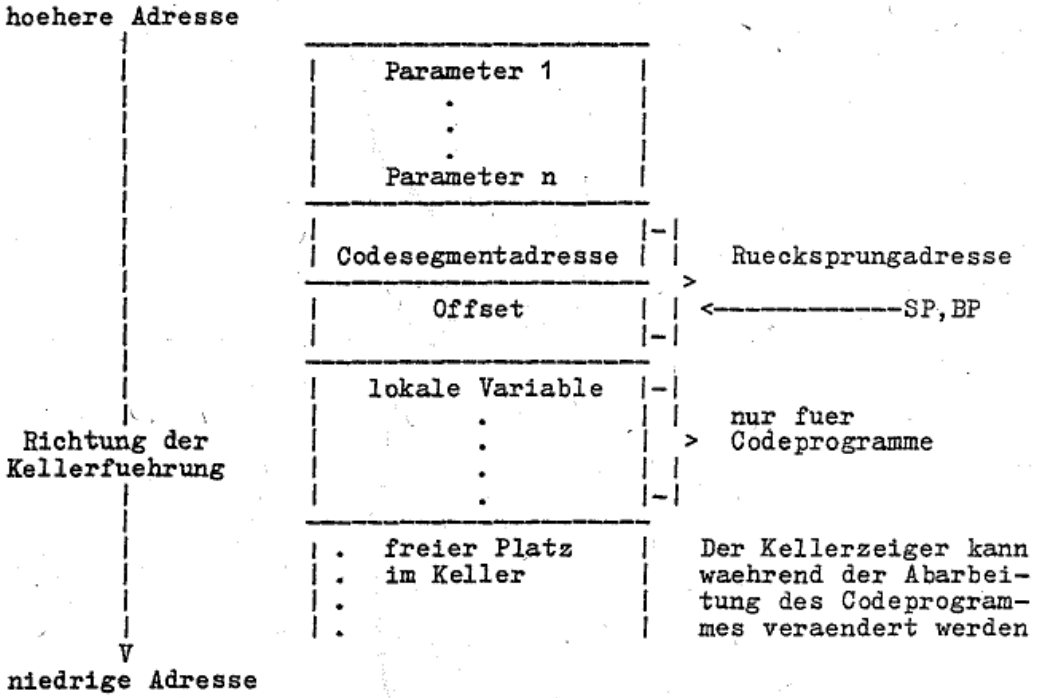


Bild 2: Kellerzustand zur Zeit der Abarbeitung des Codeprogramms

Die folgenden Regeln muessen beim Schreiben von Codeprogrammen beachtet werden:
- Ein Codeprogramm darf die Register AX, BX, CX, DX, SI, DI und BP zerstoeren.

- Ein Codeprogramm muss die Zahl und die Laenge der uebergebenen Parameter kennen. Wird angenommen, dass der aktuelle Stand des Kellerzeigers SP in das Register BP uebertragen wird (durch MOV BP,SP), so kann auf die Parameteradressen durch positive Offsets zu (BP) zugegriffen werden. Hat ein Codeprogramm zum Beispiel 3 Parameter, so kann auf die Adresse des ersten Parameters durch 8(BP), auf die Adresse des zweiten durch 6(BP) und auf die Adresse des dritten durch 4(BP) zugegriffen werden.
- Das Codeprogramm muss den Ruecksprung in das rufende Programm durch den Befehl RET <n> realisieren, wobei n die Zahl der Parameter mal 2 ist. Durch diesen Befehl wird der Keller wieder in den Zustand vor dem Ruf versetzt.
- Die Uebergabe von Werten an das BASIC-Programm erfolgt ueber Variablen aus der Argumentliste, denen im Codeprogramm die Resultatwerte zugeordnet werden.
- Ist ein Parameter eine Zeichenkette, so zeigt die Parameteradresse auf einen sogenannten "String-Deskriptor" von 3 Byte Laenge. Byte 0 enthaelt die Laenge der Zeichenkette (0 bis 255). Die Byte 1 und 2 enthalten die niederwertigen und hoeherwertigen 8 bit der Startadresse der Zeichenkette im Zeichenkettenspeicher.

  Hinweis
  ———————

  Enthaelt der Parameter eine Zeichenkettenkonstante des Programmes, so verweist der String-Deskriptor auf den Programmtext. Es ist deshalb leicht moeglich, das Programm zu aendern oder zu zerstoeren. Zur Vermeidung solcher Effekte sollte eine leere Zeichenkette an die Zeichenkettenkonstanten des Programmes angefuegt werden, zum Beispiel

  ```
  20 CALL ("BASIC"+"")
  ```

  Dies bewirkt ein Kopieren der Zeichenkettenkonstanten in den Zeichenkettenspeicher. Anschliessend kann die Zeichenkette ohne Auswirkungen auf das Programm modifiziert werden.
- Zeichenketten koennen durch das Codeprogramm veraendert werden, jedoch darf die Zeichenzahl nicht geaendert werden. BASIC kann Zeichenketten nicht korrekt hantieren, wenn externe Routinen ihre Laenge aendern.
- Interne Daten des Codeprogramms muessen bezueglich CS oder ES adressiert werden. DS darf nicht veraendert werden.

Beispiel
————————

```
100  DEF SEG=&H8000
110  CODEPROG=&H7FA
120  CALL CODEPROG(A,B¤,C)
```

In Zeile 100 wird als Segmentadresse hexadezimal 8000 festgelegt. In der Zeile 120 wird die 20-bit-Adresse des zu rufenden Codeprogramms wie folgt ermittelt:
Segmentadresse*16+Inhalt von CODEPROG
(Die Multiplikation der Segmentadresse ist eine Funktion des Prozessors, nicht von BASIC). Demzufolge wird als absolute Adresse des Codeprogramms hexadezimal 807FA ermittelt.
Die folgenden Assemblerbefehle demonstrieren den Zugriff auf Parameter und das Speichern eines Resultatwertes in die Variable C.

```
MOV     BP,SP
MOV     BX,6[BP]        ;Adresse des String-Deskriptors
MOV     CL,[BX]         ;CL=Laenge der Zeichenkette
MOV     DX,1[BX]        ;DX=Adresse der Zeichenkette
        .
        .
        .
MOV     SI,8[BP]        ;SI=Adresse von A
MOV     DI,4[BP]        ;DI=Adresse von C
MOVS    WORD            ;C:=A
RET     6
```

Hinweis

Dem gerufenen Programm muessen die Typen der Parameter bekannt sein. Im obigen Beispiel werden durch den Befehl MOVS WORD nur 2 Byte kopiert. Dies ist richtig, wenn die Variablen A und C vom Typ Integer sind. Es muessen jedoch 4 Byte kopiert werden, wenn einfache Genauigkeit vorliegt, bzw. 8 Byte, wenn doppelte Genauigkeit vorliegt.

9.1.2. USR-Funktionsruf

Syntax

USR[<ziffer>][(parameter)]

Semantik

Der USR-Funktionsruf wird zum Ruf einer Coderoutine des Nutzers (in Assemblersprache) mit einem Parameter verwendet.
Die Ziffer nach USR muss zwischen 0 und 9 liegen und korrespondiert mit der Ziffer, die in der DEF-USR-Anweisung fuer diese Routine vogesehen wurde. Ist keine Ziffer angegeben, wird USR0 angenommen.
Vor einem USR-Funktionsruf muss eine DEF-SEG-Anweisung ausgefuehrt werden, um zu sichern, dass das Codesegment zum gerufenen Codeprogramm zeigt. Die in der DEF-SEG-Anweisung angegebene Segmentadresse ist die Adresse des ersten Segmentes des Codeprogrammes.
Fuer jede USR-Funktion muss in einer korrespondierenden DEF-USR-Anweisung der Offset fuer den Ruf des Codeprogramms definiert werden. Aus diesem Offset und der aktuellen Segmentadresse wird die Startadresse des Codeprogramms bestimmt.

Parametervermittlung

Wenn ein USR-Funktionsruf ausgefuehrt wird, so enthaelt das Register AL einen Wert der den Typ des gegebenen Parameters in der folgenden Art spezifiziert:
2 Integer (2 Byte, Darstellung im Zweier-Komplement)
3 String
4 einfache Genauigkeit
8 doppelte Genauigkeit
Wenn der Parameter von numerischem Typ ist, so enthaelt BX einen Verweis zu einem Speicherbereich, in dem der Parameter wie folgt gespeichert ist:

```
niederwertige Adresse                                  hoeherwertige Adresse
   ------------------------------------------------------------------>
-------------------------------------------------------------------------
| Byte7 | Byte6 | Byte5 | Byte4 | Byte3 | Byte2 | Byte1 | Byte0 |
-------------------------------------------------------------------------
                                  ^
                                  |
                                 BX
```

Integer-Parameter

Byte2: hoeherwertige 8 bit des Parameters
Byte3: niederwertige 8 bit des Parameters

Parameter einfacher Genauigkeit

Byte0: Exponent-128, dabei steht der Binaerpunkt links neben dem ersten bit der Mantisse
Byte1: Enthaelt die hoeherwertigen sieben bit der Mantisse, an die implizit eine fuehrende 1 angefuegt werden muss. Bit 7 enthaelt das Vorzeichen (0 ist positiv, 1 ist negativ).
Byte2: Enthaelt die naechsten 8 bit der Mantisse
Byte3: Enthaelt die naechsten (niederwertigsten) 8 bit der Mantisse

Parameter doppelter Genauigkeit

Byte0-Byte3: wie oben
Byte4-Byte7: Enthaelt die weiteren bit der Mantisse (Byte 7 enthaelt die niederwertigsten bit)

Wenn der Parameter eine Zeichenkette ist, so zeigt DX auf einen 3 Byte langen String-Deskriptor, der in Byte0 die Laenge der Zeichenkette (0 bis 255) und in den Byte1 und 2 die Startadresse der Zeichenkette enthaelt (s. Abschn. 9.1.1.).

Hinweis

Enthaelt der Parameter eine Zeichenkettenkonstante des Programms, so verweist der String-Deskriptor auf den Programmtext. Es ist deshalb leicht moeglich das Programm zu aendern oder zu zerstoeren (s. Abschn. 9.1.1.).
Gewoehnlich stimmt der Typ des Resultatwertes einer USR-Funktion mit dem Typ des Parameters (integer, string, einfache oder doppelte Genauigkeit) ueberein.

9.1.3. DEF-USR-Anweisung

Syntax

DEF USR[<ziffer>]=<integer ausdruck>

Semantik

Durch die angegebene Ziffer wird die Nummer (0-9) der Assemblerroutine angegeben, deren Startadresse durch den Integer-Ausdruck definiert wird. Wird keine Ziffer angegeben, wird die Startadresse der USR0-Routine definiert.
Bestehende Definitionen fuer Startadressen koennen durch neue DEF-USR-Anweisungen ueberschrieben werden, falls weitere Unterprogramme benutzt werden sollen.

Beispiel

```
600 DEF USR0=&H8100
 .
 .
920 FC=USR0(FC)
```

9.1.4. VARPTR-Funktion

Syntax

VARPTR(<variablenname>)

Semantik

Die Funktion liefert die Adresse des ersten Byte des Datums, das durch den Variablennamen identifiziert wird.
Vor der Ausfuehrung der VARPTR-Funktion muss die Variable bereits verwendet worden sein, andernfalls wird ein Fehler gemeldet. Es koennen Variablen beliebigen Typs angegeben werden (numerische, string, Felder). Die resultierende Adresse ist ein Integer-Wert im Bereich von 32767 bis -32768. Ist die Adresse negativ, so erhaelt man die aktuelle Adresse durch Addition von 65536.
VARPTR wird zumeist benutzt, um die Adresse einer Variablen oder eines Feldes an ein Codeprogramm zu uebergeben. Ein Funktionsruf der Form VARPTR(A(0)) wird gewoehnlich verwendet, wenn ein Feld uebergeben werden soll, da so die Adresse des Anfangselementes des Feldes bereitgestellt wird.

Hinweis

Alle einfachen Variablen sollten bereits verwendet worden sein, bevor eine VARPTR-Funktion fuer ein Feld ausgefuehrt wird, weil sich die Adressen von Feldern aendern, wenn eine neue einfache Variable eingefuehrt wird.

Beispiel

```
100  X=USR(VARPTR(Y))
```

9.2. DEF-SEG-Anweisung

Syntax

```
DEF SEG[=<adresse>]
```

Semantik

Mit Hilfe der Anweisung wird die Segmentadresse fuer ein zu rufendes Codeprogramm oder fuer PEEK- und POKE-Anweisungen festgelegt. Als Adresse kann ein numerischer Ausdruck angegeben werden, der einen vorzeichenlosen ganzzahligen Wert im Bereich 0 bis 65535 liefert. Liegt die Adresse ausserhalb des angegebenen Wertebereiches, so erfolgt eine Fehlermeldung "illegal function call".
Ist keine Adresse vorgegeben, so wird als Segmentadresse die Datensegmentadresse (DS) des BASIC-Interpreters angenommen.
Zur Ermittlung der absoluten Speicheradresse wird bei der Abarbeitung von PEEK-, POKE- oder CALL-Anweisungen die durch DEF SEG eingestellte Segmentadresse um 4 Stellen nach links verschoben und anschliessend mit dem 16-bit-Offset addiert. Die Linksverschiebung der Segmentadresse ist eine Funktion des Prozessors, nicht von BASIC.
BASIC prueft nicht, ob die resultierende Adresse zulaessig ist.

Hinweis

Die Schluesselwoerter DEF und SEG muessen unbedingt durch ein Leerzeichen voneinander getrennt werden. Ansonsten wuerde BASIC die Anweisung DEFSEG=100 als einfache Wertzuweisung interpretieren (Variable DEFSEG erhaelt den Wert 100).

Beispiel

```
10 DEF SEG=&HD800
```
Als Segmentadresse wird hexadezimal D800 festgelegt (=Pufferadresse des Terminals).
```
20 DEF SEG
```
Die Segmentadresse wird auf die Datensegmentadresse des BASIC-Interpreters eingestellt.

9.3. PEEK-Funktion

Syntax

```
PEEK(I)
```

Semantik

Die PEEK-Funktion liefert einen Integer-Wert im Bereich von 0 bis 255, der aus dem Byte des Speichers mit dem Offset I gelesen wurde. Die Segmentadresse wird durch DEF SEG eingestellt.
Es muss 0<=I<=65536 gelten. PEEK ist die komplementaere Funktion zur POKE-Anweisung.

Beispiel

```
A=PEEK(&H5A00)
```

9.4. POKE-Anweisung

Syntax

POKE <integer ausdruck i>,<integer ausdruck j>

Semantik

Die POKE-Anweisung wird zum Schreiben eines Byte benutzt.
- <integer ausdruck i> liefert einen Offset im Bereich 0 bis 65536, zu dem ein Datenbyte uebertragen wird. Die Segmentadresse zu diesem Offset kann durch DEF-SEG eingestellt werden.
- <integer ausdruck j> enthaelt das zu uebertragende Datenbyte (Wertebereich 0 bis 255).

Die komplementaere Funktion zu POKE ist PEEK. PEEK hat als Argument eine Speicheradresse, on der ein Byte gelesen wird. POKE und PEEK werden zum Laden von Assemblerprogrammen und zum Datentausch verwendet.

9.5. INP-Funktion

Syntax

INP(I)

I - Integerausdruck im Bereich 0 bis 65535

Semantik

Die INP-Funktion liefert ein Datenbyte, das vom Port I gelesen wurde. INP ist die komplementaere Funktion zur OUT-Anweisung.

Beispiel

```
100 Y=INP(255)
```

9.6. OUT-Anweisung

Syntax

OUT I,J

I,J sind Integerausdruecke
Es muss 0<=I<=65535 und 0<=J<=255 gelten.

Semantik

Mit der OUT-Anweisung wird ein Datenbyte zu einem Port gesendet. I gibt die Portnummer an. J enthaelt die zu uebertragende Information.
OUT ist die komplementaere Funktion zur INP-Anweisung.

Beispiel

```
10 OUT 12345,255
```
In Assemblersprache entspricht dies der Befehlsfolge:
```
MOV DX,12345
MOV AL,255
OUT DX,AL
```

9.7. WAIT-Anweisung

Syntax

WAIT <portnummer>,I[,J]

I und J sind Integer-Ausdruecke. Es muss gelten 0<=I,J<=255.

Semantik

Die WAIT-Anweisung suspendiert die Programmabarbeitung bis der angegebene Eingabeport ein spezielles Bitmuster enthaelt. Aus dem Port wird das Bitmuster gelesen. Dieses wird mit dem Integer-Ausdruck J disjunktiv (Exklusives ODER) und anschliessend mit dem Integer-Ausdruck I konjunktiv (AND) verbunden. Solange die Verknuepfung den Wert Null liefert, wird diese Pruefroutine wiederholt. Ist das Ergebnis ungleich Null, so wird die Programmausfuehrung bei der naechsten Anweisung fortgesetzt. Fuer den Integer-Ausdruck J wird standardmaessig der Wert 0 angenommen.
Es ist moeglich, dass durch eine WAIT-Anweisung eine endlose Schleife entsteht. In diesem Fall sollte man eingreifen und einen RESTART vornehmen.

Beispiel

```
200 WAIT 32,2
```

Anlage 1: Fehlercodes und Fehlermeldungen

Code	Nr.	Ursache
NF	1	NEXT without FOR Zu einer NEXT-Anweisung laesst sich anhand der Variablen keine zuvor abgearbeitete FOR-Anweisung finden.
SN	2	Syntax error Es wird eine Zeile gefunden, die eine unkorrekte Folge von Zeichen enthaelt.
RG	3	RETURN without GOSUB Es wird eine RETURN-Anweisung gefunden, ohne dass vorher eine zugehoerige GOSUB-Anweisung ausgefuehrt wurde.
OD	4	Out of data Es soll eine READ-Anweisung ausgefuehrt werden, jedoch existieren keine DATA-Anweisungen mehr im Programm.
FC	5	Illegal function call Ein unzulaessiger Parameter wurde an eine Funktion uebergeben. Ein solcher Fehler kann folgende Ursachen haben: - negative oder zu grosse Indizes - ein Argument <=0 bei LOG - ein Argument < 0 bei SQR - eine negative Basis mit nicht ganzzahligen Exponenten bei Potenzierung - der Aufruf einer USR-Funktion, deren Startadresse nicht vorgegeben wurde - ein fehlerhaftes Argument fuer MID¤, LEFT¤, RIGHT¤, INP, OUT, WAIT, PEEK, POKE, TAB, SPC, STRING¤, SPACE¤, INSTR oder ON...GOTO
OV	6	Overflow Das Resultat einer Berechnung ueberschreitet die groesste darstellbare Zahl. Wenn ein Unterlauf eintritt, wird das Resultat auf Null gesetzt und die Programmabarbeitung ohne Fehlermeldung fortgesetzt.
OM	7	Out of memory Ein Programm ist zu lang, hat zu viele Laufanweisungen oder GOSUB-Anweisungen, zu viele Variable oder Ausdruecke, die zu kompliziert sind.
UL	8	Undefined line In einer GOTO-, GOSUB- oder IF-Anweisung bzw. einem DELETE-Kommando wird auf eine nicht existierende Zeile bezug genommen.
BS	9	Subscript out of range Ein Feldelement enthaelt entweder einen Index, der ausserhalb der Dimension des Feldes liegt, oder eine unzulaessige Zahl von Indizes.

DD 10 Redimensioned array
Es sind entweder zwei DIM-Anweisungen fuer das gleiche Feld angegeben oder es wird eine DIM-Anweisung angegeben, nachdem die Standarddimension 10 fuer das Feld festgelegt wurde.

/0 11 Division by zero
In einem Ausdruck tritt eine Division durch Null auf bzw. eine Potenzierung besitzt die Basis Null und einen negativen Exponenten. Die groesste darstellbare Zahl mit dem Vorzeichen des Zaehlers wird als Ergebnis der Division angenommen bzw. die groesste darstellbare positive Zahl wird als Ergebnis der Potenzierung angenommen und die Berechnung fortgesetzt.

ID 12 Illegal direkt
Eine Anweisung, die im direkten Modus verboten ist, wird als Kommando im direkten Modus angewandt.

TM 13 Type mismatch
Einer String-Variablen wird ein numerischer Wert zugewiesen oder umgekehrt, bzw. einer Funktion, die ein numerisches Argument besitzt, wird ein String-Argument uebergeben oder umgekehrt.

OS 14 Out of string space
Durch die Arbeit mit String-Werten wird der zur Verfuegung stehende, freie Speicher ueberschritten. Die Speicherzuweisung an Zeichenketten erfolgt dynamisch.

LS 15 String too long
Es wird versucht, eine Zeichenkette mit mehr als 255 Zeichen zu erzeugen.

ST 16 String formula too complex
Ein String-Ausdruck ist zu lang oder zu komplex. Der Ausdruck sollte in kleinere Ausdruecke aufgeteilt werden.

CN 17 Can't continue
Es wird versucht, ein Programm fortzusetzen, das
- wegen eines Fehlers unterbrochen wurde,
- nach einer Unterbrechung der Abarbeitung geaendert wurde, oder
- nicht existiert.

18 Undefined user function
Eine USR-Funktion wird aufgerufen, bevor sie definiert (DEF-Anweisung) wird.

19 No RESUME
Eine Fehlerbehandlungsroutine wird betreten, enthaelt jedoch keine RESUME-Anweisung.

20 RESUME without error
Eine RESUME-Anweisung wird erreicht, bevor eine Fehlerbehandlungsroutine aufgerufen wird.

21 Unprintable error
Fuer die vorliegende Fehlerbehandlung existiert keine Fehlermeldung. Dies tritt i.a. bei einer ERROR-Anweisung mit undefiniertem Fehlercode auf.

22 Missing operand
Ein Ausdruck enthaelt einen Operator ohne nachfolgenden Operanden.

23 Line buffer overflow
Es wird versucht eine Zeile einzugeben, die zu viele Zeichen enthaelt.

26 FOR without NEXT
Es wird eine FOR-Anweisung gefunden, zu der keine passende NEXT-Anweisung existiert.

29 WHILE without WEND
Zu einer WHILE-Anweisung existiert keine passende WEND-Anweisung.

30 WEND without WHILE
Es wird eine WEND-Anweisung gefunden, zu der keine passende WHILE-Anweisung existiert.

50 Field Overflow
In einer FIELD-Anweisung werden mehr Byte reserviert, als fuer die Direktzugriffsdatei als Datensatzlaenge spezifiziert wurde.

52 Bad file number
In einer Anweisung oder einem Kommando wird auf eine Datei zugegriffen, die entweder nicht eroeffnet ist oder deren Dateinummer nicht in dem bei der Initialisierung spezifizierten Bereich liegt.

53 File not found
Ein LOAD-Kommando oder eine KILL- bzw. OPEN-Anweisung greifen auf eine Datei zu, die auf der spezifizierten Diskette nicht vorhanden ist.

54 Bad file mode
Es wird versucht, PUT, GET oder LOF auf eine sequentielle Datei anzuwenden, mit LOAD eine Direktzugriffsdatei zu laden oder eine OPEN-Anweisung mit einem anderen Dateimodus als I, O oder R auszufuehren.

55 File already open
Es wird versucht, eine Datei, die bereits eroeffnet ist, erneut als sequentielle Ausgabedatei zu eroeffnen, bzw. eine KILL-Anweisung wird auf eine geoeffnete Datei angewandt.

57 Disk I/O error
Bei einer E/A-Operation mit der Diskette tritt ein E/A-Fehler auf.

58 File already exists
Der in einer NAME-Anweisung spezifizierte Dateiname ist identisch mit dem Namen einer Datei, die sich bereits auf der Diskette befindet.

61 Disk full
Der gesamte Speicherplatz der Diskette ist belegt.

62 Input past end
Eine INPUT-Anweisung wird ausgefuehrt, nachdem bereits alle Daten der Datei eingegeben wurden bzw. auf eine leere Datei angewandt. Zur Vermeidung dieses Fehlers kann die EOF-Funktion zum Erkennen des Dateiendes verwendet werden.

63 Bad record number
In einer PUT- oder GET-Anweisung ist die Datensatznummer entweder groesser als die maximale (32767) oder gleich Null.

64 Bad file name
In einem LOAD- oder SAVE-Kommando bzw. einer KILL- oder OPEN-Anweisung wird ein Dateiname verwendet, der fehlerhaft aufgebaut ist (z.B. zu viele Zeichen).

66 Direct statement in file
Waehrend des Ladens einer Datei im externen Textformat mit dem Kommando LOAD wird in dieser Datei eine Anweisung im direkten Modus gefunden. Das Laden wird abgebrochen.

67 Too many files
Es wird versucht, eine neue Datei zu erstellen, obwohl alle 255 Dateiverzeichniseintragungen belegt sind.

Anlage 2: Syntaxzusammenstellung

Kommandos

```
AUTO [<zeilennummer>[,[<schrittweite>]]]

DELETE [<zeilennummer>][-<zeilennummer>]

EDIT <zeilennummer>

oder

EDIT .

LIST[<zeilennummer>][-[<zeilennummer>]]

LLIST [<zeilennummer>][-[<zeilennummer>]]

RENUM [<zeilennummer neu>][,[<zeilennummer alt>]
      [,<schrittweite>]]

NEW

CLEAR [,[<speicherende>][,<stack>]]

CONT

RUN [<zeilennummer>]

oder

RUN <dateispezifikation>[,R]

TRON

TROFF

SYSTEM

FILES [<dateispezifikation>]

KILL <dateispezifikation>
```

```
LOAD <dateispezifikation>[,R]

MERGE <dateispezifikation>

NAME <dateispezifikation alte Datei> AS <dateispezifikation
      neue Datei>

SAVE <dateispezifikation>[,<option>]

     Fuer Option kann A oder P stehen.

RESET
```

Anweisungen

```
REM <text>

oder

' <text>

DEF<typ> <bereichsangabe> [,<bereichsangabe>... ]

   <typ>::= INT,SNG,DBL,STR

[LET] <variable> = <ausdruck>

SWAP <variable>, <variable>

MID¤ (<zeichenkettenausdruck 1>,<n>[,<m>]) =
     <zeichenkettenausdruck 2>

     <n> und <m> sind Integerausdruecke

RANDOMIZE [<numerischer ausdruck>]

INPUT [;][<"ausschrift">;]<variablenliste>

LINE INPUT [;][<"ausschrift">];<zeichenkettenvariable>

PRINT [<liste von ausdruecken>]    oder
  ?   [<liste von ausdruecken>]

PRINT USING <formatbeschreibung>;<liste von ausdruecken>
```

```
LPRINT [<liste von ausdruecken>]

LPRINT USING <formatbeschreibung>;<liste von ausdruecken>

WRITE [<liste von ausdruecken>]

WIDTH <numerischer ausdruck>

DATA <liste von ausdruecken>

READ <liste von variablen>

RESTORE [<zeilennummer>]

END

STOP

GOTO <zeilennummer>

GOSUB <zeilennummer>

ON <ausdruck> GOTO <liste von zeilennummern>

ON <ausdruck> GOSUB <liste von zeilennummern>

FOR <variable>= <a> TO <e> [STEP <s>]
 .
 .
[<anweisungsfolge>]
 .
 .
 .
NEXT [<variable>]

     <a>,<e>,<s>  sind numerische Ausdruecke
     Variable  einfach genaue Variable oder Intergervariable

WHILE <ausdruck>
  .
  .
[<anweisungen>]
  .
  .
WEND
```

```
IF <ausdruck> THEN <anweisung(en)>/<zeilennummer>
   [ELSE<anweisun(en)>/<zeilennummer>]

IF <ausdruck> GOTO <zeilennummer>
   [ELSE<anweisung(en)>/<zeilennummer>

DIM <liste indizierter variablen>

OPTION BASE 0

OPTION BASE 1

DEF FN<funktionsname>[(<parameterliste>)]=<ausdruck>

CHAIN [MERGE]<dateiname>[,[<zeile>][,ALL][,DELETE<bereich>]]

COMMON <variablenliste>

ERROR <integerausdruck>

ON ERROR GOTO <zeilennummer>

RESUME

RESUME 0

RESUME NEXT

RESUME <zeilennummer>
```

Anweisungen und Funktionen fuer die Dateiarbeit

Eroeffnen und Schliessen von Dateien

```
OPEN <dateimodus 1>,[#]<dateinummer>,<dateispezifikation>
     [,<satzlaenge>]

CLOSE[[#]<dateinummer>[,[#]<dateinummer>... ]]
```

E/A mit sequentiellen Dateien

```
PRINT#<dateinummer>,[USING<zeichenkettenausdruck>;]
      <liste von ausdruecken>
```

WRITE#<dateinummer>,<liste von ausdruecken>

INPUT#<dateinummer>,<variablenliste>

LINE INPUT#<dateinummer>,<zeichenkettenvariable>

E/A mit Direktzugriffsdateien

FIELD[#]<dateinummer>,<feldlaenge> AS <stringvariable>
[,<feldlaenge> AS <stringvariable>]....

LSET <zeichenkettenvariable>=<zeichenkettenausdruck>

RSET <zeichenkettenvariable>=<zeichenkettenausdruck>

PUT[#]<dateinummer>[,<satznummer>]

GET[#]<dateinummer>[,<satznummer>]

MKI¤(<integer-ausdruck>)

MKS¤(<einfach genauer ausdruck>)

MKD¤(<doppelt genauer ausdruck>)

CVI(<zeichenkette aus 2 Zeichen>)

CVS(<zeichenkette aus 4 Zeichen>)

CVD(<zeichenkette aus 8 Zeichen>)

Funktionen

EOF(<dateinummer>)

LOC(<dateinummer>)

LOF (<dateinummer>)

INPUT¤(X[,[#]Y])

VARPTR(#<dateinummer>)

Rechnerspezifische Ausdrucksmittel

CALL <variablenname>[(<argumentliste>)]

USR[<ziffer>][(parameter)]

DEF USR[<ziffer>]=<integer ausdruck>

VARPTR(<variablenname>)

DEF SEG[=<adresse>]

PEEK(I)

POKE <integer ausdruck i>,<integer ausdruck j>

INP(I)

I - Integerausdruck im Bereich 0 bis 65535

OUT I,J

I,J sind Integerausdruecke
Es muss 0<=I<=65535 und 0<=J<=255 gelten.

WAIT <portnummer>,I[,J]

I und J sind Integer-Ausdruecke. Es muss gelten 0<=I,J<=255.

Anlage 3: Reservierte Woerter

Die folgende Liste enthaelt alle fuer BASIC-1700 reservierten Woerter.

ABS
AND
ASC
ATN
AUTO
CALL
CDBL
CHAIN
CHR¤
CINT
CLEAR
CLOSE
COMMON
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
EDIT
ELSE
END
EOF
EQV
ERASE
ERL
ERR
ERROR
EXP
FIELD
FIX
FN
FOR
FRE
GET
GOSUB
HEX¤
IF
IMP
INKEY¤
INP
INPUT
INPUT#
INPUT¤
INSTR
INT
KILL
LEFT¤
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOF
LOG
LPOS
LPRINT
LPRINT#
LSET
MERGE
MID¤
MKD¤
MKI¤
MKS¤
MOD
NAME
NEW
NEXT
NOT
OCT¤
ON
OPEN
OPTION
OR
OUT
PEEK
POKE
POS
PRINT
PRINT#
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME
RETURN
RIGTH¤
RND
RSET
RUN
SAVE
SGN
SIN
SPACE¤
SPC
SQR
STEP
STOP
STR¤
STRING¤
SWAP
SYSTEM
TAB
TAN
THEN
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR#
WAIT
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

Anlage 4: Beispiel fuer die Arbeit mit Direktzugriffsdateien

Das folgende Programm demonstriert die Arbeit mit Direktzugriffsdateien am Beispiel eines Telefonverzeichnisses. Mit diesem Verzeichnis sind die Operationen Erweitern, Loeschen, Sortieren, Verdichten, Suchen, Ausdrucken und Terminalausgabe moeglich. Fuer jede Operation ist ein Teilprogramm zustaendig. Die Steuerung der Teilprogramme erfolgt ueber ein Menue. Die Funktionen zum Lesen und Schreiben von Saetzen und zur Kennzeichenverarbeitung enthalten einen gemeinsam von allen Teilen benutzten Wurzelmodul.

Programmteil	Funktion
TINI.BAS	Aufruf eines leeren Telefonverzeichnisses
PHONE.BAS	Wurzelmodul
TMENUE.BAS	Menuesteuerung
TEXTND.BAS	Erweitern des Telefonverzeichnisses
TDEL.BAS	Loeschen von Eintragungen
TSORT.BAS	Sortieren des Verzeichnisses
TCOMP.BAS	Verdichten
TSEAR.BAS	Schnelle Suche nach Eintragungen
TLIST.BAS	Auslisten des Verzeichnisses
TPRINT.BAS	Ausdrucken des Verzeichnisses

Programmteil TINI.BAS

```
10 REM ---------- TELEPHONE NUMBERS ----------
20 REM
30 REM INITIALIZATION PART
40 OPEN "R",1,"PHONE.DAT",80
50 FIELD 1,2 AS RECORDS¤, 9 AS DATE¤, 2 AS SORT¤, 2 AS COMP¤
60 LINE INPUT "DATE:  (DD-MMM-YY) :",DAT¤
70 LSET DATE¤ = DAT¤
80 LSET RECORDS¤=MKI¤(0)
90 LSET SORT¤ = MKI¤(1)
100 LSET COMP¤ = MKI¤(1)
110 PUT #1,1
120 CLOSE
```

Programmteil PHONE.BAS

```
10 CHAIN MERGE "TMENUE" ,2000,ALL
20 REM ---------- SUBFUNCTIONS ----------
30 REM ---------- OPEN AND READ RECORD 0 ----------
40 OPEN "R",1,"PHONE.DAT",80
50 FIELD 1,2 AS RECORDS¤,9 AS DATE¤,2 AS SORT¤, 2 AS COMP¤
60 GET #1,1
70 PRINT "LAST REVISION DATE WAS :";DATE¤
80 PRINT CVI(RECORDS¤);"RECORDS IN FILE PHONE.DAT"
90 PRINT
100 T.RECS = CVI(RECORDS¤): T.SORT=CVI(SORT¤) :T.COMP=CVI(COMP¤)
110 T.DAT¤ = DATE¤
120 RETURN
130 REM ----------CLOSE AND WRITE RECORD 0 ----------
140 FIELD 1,2 AS RECORDS¤,9 AS DATE¤,2 AS SORT¤, 2 AS COMP¤
```

```
150 LSET RECORDS¤ = MKI¤(T.RECS) : LSET SORT¤ = MKI¤(T.SORT)
160 LSET DATE¤ = T.DAT¤ : LSET COMP¤ = MKI¤(T.COMP)
170 PUT #1,1
180 CLOSE
190 RETURN
200 REM ----------READ ONE RECORD ----------
210 FIELD 1,30 AS LASTNAME¤,30 AS FIRSTNAME¤,8 AS TNUM¤
220 GET #1,T.RECN+1
230 T.LASTN¤ = LASTNAME¤
240 T.FIRSTN¤ = FIRSTNAME¤
250 T.TNUM# = CVD(TNUM¤)
260 RETURN
270 REM ----------WRITE ONE RECORD ----------
280 FIELD 1,30 AS LASTNAME¤,30 AS FIRSTNAME¤,8 AS TNUM¤
290 LSET LASTNAME¤ = T.LASTN¤
300 LSET FIRSTNAME¤ = T.FIRSTN¤
310 LSET TNUM¤ = MKD¤(T.TNUM#)
320 PUT #1,T.RECN+1
330 RETURN
```

Programmteil TMENUE.BAS

```
2000 PRINT CHR¤(27);"[2J"
2010 PRINT "FOLLOWING FUNCTIONS ARE AVAILABLE"
2020 PRINT "----------------------------------"
2030 PRINT
2040 PRINT "(1) EXTEND "
2050 PRINT "(2) LIST"
2060 PRINT "(3) PRINT"
2070 PRINT "(4) DELETE"
2080 PRINT "(5) SORT"
2090 PRINT "(6) COMPRESS"
2100 PRINT "(7) SEARCH"
2110 INPUT "FUNCTION NUMBER :",I
2120 IF I > 7 OR I < 1 THEN GOTO 2000
2130 ON I GOTO 2140,2150,2160,2170,2180,2190,2200
2140 CHAIN MERGE "TEXTND",2000,ALL,DELETE 2000-9999
2150 CHAIN MERGE "TLIST" ,2000,ALL,DELETE 2000-9999
2160 CHAIN MERGE "TPRINT",2000,ALL,DELETE 2000-9999
2170 CHAIN MERGE "TDEL"  ,2000,ALL,DELETE 2000-9999
2180 CHAIN MERGE "TSORT" ,2000,ALL,DELETE 2000-9999
2190 CHAIN MERGE "TCOMP" ,2000,ALL,DELETE 2000-9999
2200 CHAIN MERGE "TSEAR" ,2000,ALL,DELETE 2000-9999
9999 REM END OF SEGMENT
```

Programmteil TEXTND.BAS

```
2000 GOSUB 30 'OPEN AND READ RECORD 0
2010 LINE INPUT "LASTNAME,FISRTNAME: ";A¤
2020 IF A¤ = "" THEN 2120
2030 P = INSTR (A¤,",")
2040 IF P = 0 THEN GOTO 2010
2050 T.LASTN¤ = MID¤(A¤,1,P-1)
2060 T.FIRSTN¤= MID¤(A¤,P+1)
2070 T.RECS = T.RECS +1
2080 T.RECN = T.RECS
2090 INPUT "NUMBER: ",T.TNUM#
2100 GOSUB 270 'WRITE RECORD
```

```
2110 GOTO 2010
2120 LINE INPUT "DATE (DD-MMM-YY) : ";T.DAT¤
2130 T.SORT = 0
2140 GOSUB 130 'WRITE RECORD 0
2150 GOTO 10 'BACK TO MENUE
2160 REM END OF SEGMENT
```

Programmteil TDEL.BAS

```
2000 '----- SEARCH A MEMBER IN TELEPHONE MAP WITH BINARY SEARCH
2010 GOSUB 30 ' READ RECORD 0
2020 IF T.RECS = 0 THEN PRINT "FILE IS EMPTY" : GOTO 2430
2030 IF T.SORT = 0 THEN PRINT "FILE MUST BE SORTED" : GOTO 2430
2040 IF T.COMP = 0 THEN PRINT "FILE MUST BE COMPRESSED":GOTO 2430
2050 REM -------BINARY SEARCH ----------
2060 LINE INPUT "LASTNAME[,FIRSTNAME] : ";MATCHNAM¤
2070 IF MATCHNAM¤ = "" THEN GOTO 2430
2080 LOWER% = 1 : UPPER% = T.RECS
2090 MID% = LOWER% + (UPPER% - LOWER%)/2
2100 IF LOWER% > UPPER% THEN PRINT "NOT FOUND" : GOTO 2410
2110 T.RECN = MID%
2130 GOSUB 200 'READ A RECORD
2140 REM --- MATCH THE NAMES ---
2150 L = INSTR(T.LASTN¤," ")-1
2160 P = INSTR(MATCHNAM¤,",")-1
2170 IF L = -1 THEN L =30
2180 IF P = -1 THEN P = LEN(MATCHNAM¤)
2190 IF MID¤(T.LASTN¤,1,L) < MID¤(MATCHNAM¤,1,P) THEN GOTO 2360
2200 IF MID¤(T.LASTN¤,1,L) > MID¤(MATCHNAM¤,1,P) THEN GOTO 2310
2210 REM --- MATCH FIRSTNAME --
2220 P = INSTR(MATCHNAM¤,",")
2230 IF P = 0 THEN GOTO 2290
2240 P = P +1
2250 L = INSTR(T.FIRSTN¤," ")-1
2260 IF L = -1 THEN L = 30
2270 IF MID¤(T.FIRSTN¤,1,L) < MID¤(MATCHNAM¤,P) THEN GOTO 2360
2280 IF MID¤(T.FIRSTN¤,1,L) > MID¤(MATCHNAM¤,P) THEN GOTO 2310
2290 ' RECORD WAS FOUND
2300 PRINT T.LASTN¤;T.FIRSTN¤;T.TNUM#: GOTO 2410
2310 ' TAKE PART FROM LOWER BOUND TO MID
2320 UPPER% = MID% -1
2330 IF LOWER% = MID% THEN PRINT "NOT FOUND" : GOTO 2410
2340 IF MID%-LOWER% = 1 THEN MID% = LOWER% : GOTO 2100
2350 GOTO 2090
2360 'TAKE PART FROM MID TO UPPER BOUND
2370 LOWER% = MID% + 1
2380 IF UPPER% = MID% THEN PRINT "NOT FOUND"  : GOTO 2410
2390 IF UPPER%-MID% = 1 THEN MID% = UPPER% :GOTO 2100
2400 GOTO 2090
2410 LINE INPUT "DELETE THIS RECORD (Y/N) ? ",A¤
2420 IF A¤ = "Y" THEN T.LASTN¤="["+T.LASTN¤ : GOSUB 270
2425 T.COMP=0 'RECORD BACK
2430 GOTO 2060
2440 GOSUB 130
2450 GOTO 10
2460 END
```

Programmteil TSORT.BAS

```
2000 '------- SORTING THE TELPHONE MAP --------
2010 GOSUB 30 'READ RECORD 0
2020 DIM LAST¤(T.RECS),FIRST¤(T.RECS),NUM#(T.RECS)
2030 FOR I = 1 TO T.RECS
2040 T.RECN=I
2050 GOSUB 200 'READ ONE RECORD
2060 LAST¤(I)=T.LASTN¤
2070 FIRST¤(I) = T.FIRSTN¤
2080 NUM#(I) = T.TNUM#
2090 NEXT
2100 FOR I = 1 TO T.RECS
2110 FOR J = I+1 TO T.RECS
2120 IF LAST¤(I)+FIRST¤(I) <= LAST¤(J)+FIRST¤(J) THEN 2160
2130 SWAP LAST¤(I),LAST¤(J)
2140 SWAP FIRST¤(I), FIRST¤(J)
2150 SWAP NUM#(I),NUM#(J)
2160 NEXT
2170 NEXT
2180 FOR I = 1 TO T.RECS
2190 T.RECN=I
2200 T.LASTN¤ = LAST¤(I)
2210 T.FIRSTN¤ = FIRST¤(I)
2220 T.TNUM# = NUM#(I)
2230 GOSUB 270
2240 NEXT
2250 T.SORT = 1
2260 GOSUB 130
2270 ERASE LAST¤,FIRST¤,NUM#
2280 GOTO 10
2290 END
```

Programmteil TCOMP.BAS

```
2000 '----- COMPRESS ------
2010 GOSUB 30 'READ RECORD 0
2020 J = 1
2030 FOR I = 1 TO T.RECS
2040 T.RECN=I
2050 GOSUB 200 ' READ ONE RECORD
2060 IF LEFT¤(T.LASTN¤,1) = "[" THEN GOTO 2100
2070 T.RECN = J
2080 IF T.RECN <> I THEN GOSUB 270 'WRITE THEN RECORD
2090 J = J + 1
2100 NEXT
2110 T.RECS= J-1
2120 T.COMP = 1
2130 GOSUB 130
2140 GOTO 10
2150 END
```

Programmteil TSEAR.BAS

```
2000 '----- SEARCH A MEMBER IN TELEPHONE MAP WITH BINARY SEARCH
2010 GOSUB 30 ' READ RECORD 0
2020 IF T.RECS = 0 THEN PRINT "FILE IS EMPTY" : GOTO 2430
2030 IF T.SORT = 0 THEN PRINT "FILE MUST BE SORTED" : GOTO 2430
2040 IF T.COMP = 0 THEN PRINT "FILE MUST BE COMPRESSED":GOTO 2430
2050 REM ------ BINARY SEARCH ----------
2060 LINE INPUT "LASTNAME[,FIRSTNAME] : ";MATCHNAM¤
2070 IF MATCHNAM¤ = "" THEN GOTO 2430
2080 LOWER% = 1 : UPPER% = T.RECS
2090 MID% = LOWER% + (UPPER% - LOWER%)/2
2100 IF LOWER% > UPPER% THEN PRINT "NOT FOUND" : GOTO 2410
2110 T.RECN = MID%
2130 GOSUB 200 'READ A RECORD
2140 REM --- MATCH THE NAMES ---
2150 L = INSTR(T.LASTN¤," ")-1
2160 P = INSTR(MATCHNAM¤,",")-1
2170 IF L = -1 THEN L =30
2180 IF P = -1 THEN P = LEN(MATCHNAM¤)
2190 IF MID¤(T.LASTN¤,1,L) < MID¤(MATCHNAM¤,1,P) THEN GOTO 2360
2200 IF MID¤(T.LASTN¤,1,L) > MID¤(MATCHNAM¤,1,P) THEN GOTO 2310
2210 REM --- MATCH FIRSTNAME ---
2220 P = INSTR(MATCHNAM¤,",")
2230 IF P = 0 THEN GOTO 2290
2240 P = P +1
2250 L = INSTR(T.FIRSTN¤," ")-1
2260 IF L = -1 THEN L = 30
2270 IF MID¤(T.FIRSTN¤,1,L) < MID¤(MATCHNAM¤,P) THEN GOTO 2360
2280 IF MID¤(T.FIRSTN¤,1,L) > MID¤(MATCHNAM¤,P) THEN GOTO 2310
2290 ' RECORD WAS FOUND
2300 PRINT T.LASTN¤;T.FIRSTN¤;T.TNUM#: GOTO 2410
2310 ' TAKE PART FROM LOWER BOUND TO MID
2320 UPPER% = MID% -1
2350 GOTO 2090
2360 'TAKE PART FROM MID TO UPPER BOUND
2370 LOWER% = MID% + 1
2400 GOTO 2090
2410 PRINT
2420 GOTO 2050
2430 GOSUB 180
2440 GOTO 10
2450 END
```

Programmteil TLIST.BAS

```
2000 '-------- LIST FUNCTION --------
2010 GOSUB 30 'GET RECORD 0 AND OPEN
2020 FOR I = 1 TO T.RECS
2030 T.RECN = I
2040 GOSUB 200 'READ ONE RECORD
2050 IF MID¤(T.LASTN¤,1,1)="[" THEN 2070
2060 PRINT T.LASTN¤;T.FIRSTN¤;T.TNUM#
2070 NEXT
2080 GOSUB 180
2090 PRINT
2100 LINE INPUT "TYPE <CR> TO CONTINUE !";A¤
2110 GOTO 10
2120 END
```

Programmteil TPRINT.BAS

```
2000 '-------- LIST FUNCTION --------
2010 GOSUB 30 'GET RECORD 0 AND OPEN
2020 FOR I = 1 TO T.RECS
2030 T.RECN = I
2040 GOSUB 200 'READ ONE RECORD
2050 IF MID¤(T.LASTN¤,1,1)="[" THEN 2070
2060 LPRINT T.LASTN¤;T.FIRSTN¤;T.TNUM#
2070 NEXT
2080 GOSUB 180
2090 PRINT
2100 LINE INPUT "TYPE <CR> TO CONTINUE !";A¤
2110 GOTO 10
2120 END
```

Anlage 5: Unterschiede zwischen BASIC-1700 und BASIC fuer SCPX 1520

Zwischen beiden Sprachversionen gibt es geringe Unterschiede, die fuer den Transfer von BASIC-Programmen zu beachten sind und die folgende Anweisungen betreffen:

- DEF-SEG-Anweisung (s. Abschn. 9.2.):
  Diese Anweisung existiert in BASIC fuer SCPX 1520 nicht. In BASIC-1700 muss sie nur dann notiert werden, wenn bei einer PEEK-, POKE-, CALL- oder USR-Anweisung Adressen angesprochen werden sollen, die ausserhalb des fuer BASIC reservierten Speicherbereiches liegen (siehe Abschnitte 9.1.1., 9.1.2., 9.3. und 9.4.).
- Behandlung numerischer Fehler:
  Die Fehler Zahlenueberlauf, Zahlenunterlauf und Division durch Null koennen in BASIC 1520 durch eine ON-ERROR-Anweisung behandelt werden, bei BASIC-1700 wirkt bei diesen Fehlern die Standardfehlerbehandlung.
- FILES-Kommando:
  Dieses Kommando ist nur fuer BASIC SCPX 1520 verfuegbar. In BASIC-1700 nicht implementiert.
- Loeschen des Bildschirms und Kursorpositionierung:
  Fuer diese Operationen sind folgende Zeichenreihen auszugeben, wobei LINE% die Bildschirmzeile (1...24) und COLUMN% die Bildschirmspalte (1...80) bei der Kursorpositionierung bezeichnet.

BASIC SCPX 1520

Bildschirm loeschen
```
CHR¤(12)
```

Kursorpositionierung
```
CHR¤(27)+CHR¤(128*LINE%)+
         CHR¤(128+COLUMN%)
```

BASIC-1700

Bildschirm loeschen
```
CHR¤(27)+"[2]"
```

Kursorpositionierung
Zum Erzeugen der auszugebenden Steuerzeichenkette definiert man folgende Funktionen:

```
DEF FNP¤(LINE%,COLUMN%)=CHR¤(27)+"["
+FNN¤(LINE%)+";"+FNN¤(COLUMN%)+"H"

DEF FNN¤(N%)=MID¤(STR¤(N%),2)
```

Neben diesen Funktionen koennen bei BASIC-1700 weitere ESCAPE Folgen zur Bildschirmsteuerung verwendet werden, die aus der Hardwaredokumentation zu entnehmen sind.

Sachwortverzeichnis

	Seite
AUTO	24
Anwenderfunktionen	23
Ausdruecke	18
BASIC-1700-Zeichensatz	11
Bediengeraet	9
Bitmuster	22
Bytetest	22
CALL	101
CHAIN	64
CLEAR	30
CLOSE	84
COMMON	65
CONT	31
CVD	95
CVI	95
CVS	95
Codeprogramm	100,102
Codeunterprogramme	15
DEF FN	63
DEFUSR	106
DELETE	25
Dateibeschreibungssbloecke	8
Deklarationszeichen	14
Direktzugriffsdatei	33,82,91,96
Diskettendatei	9
EDIT	25,36
EOF	89,97
ERASE	63
ERL	66
ERR	66
F-Option, dateispezifikation	8
FIELD	92
Fehlerbehandlung	66
Fehlercode	66
Fehlermeldungen	9
Feld	61
Felder	15
Funktion, nutzerdefinierte	14
GET	94
Ganzzahlige Konstanten	12
Hexadezimalkonstanten	13
INP	108
INPUT#	87
INPUT¤	98
Initialisierung	7
KILL	33

KOI-7-Code 23
Kommandoniveau 9
Konstanten 12

LINE-INPUT# 88
LIST 28,36
LLIST 29
LOAD 34,36
LOC 89,97
LSET 92
Logische Operatoren 21

M-Option 8
MERGE 34
MKD$ 94
MKI$ 94
MKS$ 94

NAME 35
NEW 30
Numerische Konstanten 12

ON-ERROR-GOTO 66,67
ON... GOSUB 55
ON... GOTO 55
OPEN 83
OPTION BASE 62
OUT 109
Oktalkonstanten 13
Operanden, logische Ausdruecke 18
Operatoren 18

PEEK 108
POKE 108
PRINT# 85
PRINT# USING 85
PUT 93
Programmzeile, indirekte Modus 9

RESUME 67
RSET 92
RUN 36
Reelle Konstanten 13
Reservierte Woerter 14

S-Option 8
SAVE 36
STOP 31
Spezialzeichen 11
Standardfunktionen 23
Steuerzeichen 12
String-Ausdruecke, numerische
Ausdruecke 18
String-Vergleich 23
String-Wert 23
System 33

TROFF 32
Typ-Konvertierung 16
Typdeklarationszeichen 14

USR-Funktion 104
Unterprogrammaufruf 16

VARPTR 98,106
Variable 14,92
Variablenart 15
Variablennamen 14
Vergleichsoperatoren, funktionale Operatoren 20

WAIT 109
WHILE...WEND 59
WRITE# 86

Zeichenkette 85,87,103
Zeichenkettenkonstanten 12
Zeichenkettenoperator, arithmetische Operatoren 18
Zeilennummer 9,24,25,28,29,34,55
Ziffern, alphabetische Zeichen 11

NOTIZEN

NOTIZEN

NOTIZEN

NOTIZEN